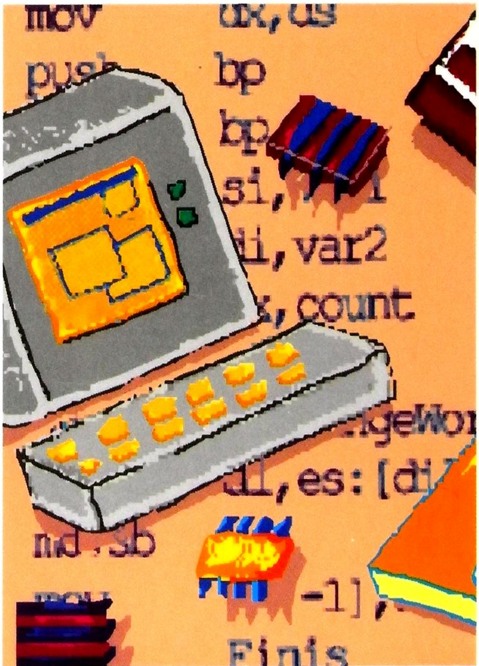
REFERENCE GUIDE

TURBO ASSEMBLER®



BORLAND

TURBO ASSEMBLER®

Borland's No-Nonsense License Statement!

This software is protected by both United States copyright law and international treaty provisions. Therefore, you must treat this software *just like a book*, with the following single exception. Borland International authorizes you to make archival copies of the software for the sole purpose of backing-up our software and protecting your investment from loss.

By saying, "just like a book," Borland means, for example, that this software may be used by any number of people and may be freely moved from one computer location to another, so long as there is **no possibility** of it being used at one location while it's being used at another. Just like a book that can't be read by two different people in two different places at the same time, neither can the software be used by two different people in two different places at the same time (unless, of course, Borland's copyright has been violated).

Programs that you write and compile using Turbo Assembler software may be used, given away or sold without additional license or fees, as long as all copies of such programs bear a copyright notice. By "copyright notice" we mean either your own copyright notice or, if you prefer, the statement, "Created using Turbo Assembler, Copyright © Borland 1988."

The sample programs included on the Turbo Assembler diskettes provide a demonstration of how to use the various features of Turbo Assembler. They are intended for educational purposes only. Borland International grants you (the registered owner of Turbo Assembler) the right to edit or modify these sample programs for your own use, but you may not give away or sell them, alone or as part of any program, in executable, object or source code form. You may, however, incorporate miscellaneous sample program routines into your programs, as long as your resulting programs do not substantially duplicate all or part of a sample program in appearance or functionality and all copies of such programs bear a copyright notice.

Limited Warranty

With respect to the physical diskette and physical documentation enclosed herein, Borland International, Inc. ("Borland") warrants the same to be free of defects in materials and workmanship for a period of 60 days from the date of purchase. In the event of notification within the warranty period of defects in material or workmanship, Borland will replace the defective diskette or documentation. **If you need to return a product, call the Borland Customer Service Department to obtain a return authorization number.** The remedy for breach of this warranty shall be limited to replacement and shall not encompass any other damages, including but not limited to loss of profit, and special, incidental, consequential, or other similar claims.

Borland International, Inc. specifically **disclaims** all other warranties, expressed or implied, including but not limited to implied warranties of merchantability and fitness for a particular purpose with respect to defects in the diskette and documentation, and the program license granted herein in particular, and without limiting operation of the program license with respect to any particular application, use, or purpose. **In no event shall Borland be liable for any loss of profit or any other commercial damage, including but not limited to special, incidental, consequential or other damages.**

Governing Law

This statement shall be construed, interpreted, and governed by the laws of the state of California. Use, duplication, or disclosure by the U.S. Government of the computer software and documentation in this package shall be subject to the restricted rights under DFARS 52.227-7013 applicable to commercial computer software.

Turbo Assembler®

リファレンスガイド

バージョン1.0

(NEC PC-9801シリーズ)
(IBM PC, XT, AT, PS/2および完全な互換機)

Copyright © 1988
All Right Reserved

Borland International
1800 Green Hills Road
P.O. Box 660001
Scotts Valley, CA 95066-0001

編訳 (株)ボーランド ジャパン 1989

本マニュアルの一部または全部を許可なく複製または転載することはできません。

目　次

はじめに……1

必要なハードウェアとソフトウェア……1

マニュアルについて……2

マニュアルの表記法……3

謝辞……4

製品に関するお問い合せ……5

第1章　定義済みシンボル……7

@code……9

@CodeSize……9

@Cpu……10

@curseg……11

@data……11

@DataSize……12

??Date……12

@fardata……13

@fardata?……13

@FileName……14

??Filename……14

??Time……15

??Version……15

@WordSize……16

第2章　演算子……17

算術演算の精度……18

演算子の優先順位……18

()……20

*……21

+(二項)……22

+(単項)……22

−(二項)……23

−(単項)……24

.……25

/……26

:……27

?……28

[]　演算子……29

AND……30

BYTE……31

DUP……32

DWORD……33

EQ……34

FAR……35

FWORD……36

GE……37

GT……38

HIGH……39

LARGE……40

LE ……41
LENGTH ……42
LOW ……43
LT ……44
MASK ……45
MOD ……46
NE ……47
NEAR ……48
NOT ……49
OFFSET ……50
OR ……51
PROC ……52
PTR ……53
PWORD ……54
QWORD ……55
SEG ……56
SHL ……57
SHORT ……58
SHR ……59
SIZE ……60
SMALL ……61
SYMTYPE ……62
TBYTE ……62
THIS ……63
.TYPE ……64
TYPE ……65
UNKNOWN ……66
WIDTH ……68
WORD ……69
XOR ……70
特別のマクロ演算子 ……71
& ……72
<> ……73
! ……74
% ……75
;; ……76

第3章　疑似命令 ……77

サンプル疑似命令 ……78
.186 ……79
.286 ……80
.286C ……81
.286P ……81
.287 ……82
.386 ……83
.386C ……83
.386P ……84
.387 ……85
.8086 ……86
.8087 ……87
: ……88
= ……89
ALIGN ……90
.ALPHA ……91
ARG ……92
ASSUME ……95
%BIN ……96
CATSTR ……97
.CODE ……98
CODESEG ……99
COMM ……100
COMMENT ……102
%CONDS ……103
.CONST ……104
CONST ……104
.CREF ……105

%CREF ……105
%CREFALL ……106
%CREFREF ……107
%CREFUREF ……108
%CTLS ……109
.DATA ……110
.DATA? ……111
DATASEG ……112
DB ……113
DD ……114
%DEPTH ……116
DF ……117
DISPLAY ……119
DOSSEG ……120
DP ……121
DQ ……121
DT ……122
DW ……123
ELSE ……124
ELSEIF ……125
EMUL ……126
END ……127
ENDIF ……128
ENDM ……129
ENDP ……130
ENDS ……131
EQU ……132
.ERR ……133
ERR ……134
.ERR1 ……135
.ERR2 ……136
.ERRB ……137
.ERRDEF ……138
.ERRDIF ……139
.ERRDIFI ……140
.ERRE ……141
.ERRIDN ……142
.ERRIDNI ……143
ERRIF ……144
ERRIF1 ……144
ERRIF2 ……144
ERRIFB ……145
ERRIFDEF ……145
ERRIFDIF ……145
ERRIFDIFI ……146
ERRIFE ……146
ERRIFIDN ……146
ERRIFIDNI ……147
ERRIFNB ……147
ERRIFNDEF ……147
.ERRNB ……148
.ERRNDEF ……149
.ERRNZ ……150
EVEN ……151
EVENDATA ……152
EXITM ……153
EXTRN ……154
.FARDATA ……156
.FARDATA? ……157
FARDATA ……158
GLOBAL ……159
GROUP ……161
IDEAL ……163
IF ……164
IF1 ……165
IF2 ……166
IFB ……167
IFDEF ……168
IFDIF, IFDIFI ……169
IFE ……170

IFIDN, IFIDNI ……171
IFNB ……172
IFNDEF ……173
%INCL ……174
INCLUDE ……175
INCLUDELIB ……176
INSTR ……177
IRP ……178
IRPC ……179
JUMPS ……180
LABEL ……181
.LALL ……182
.LFCOND ……182
%LINUM ……183
%LIST ……184
.LIST ……184
LOCAL ……185
LOCALS ……187
MACRO ……189
%MACS ……190
MASM ……191
MASM51 ……192
.MODEL ……193
MODEL ……197
MULTERRS ……198
NAME ……199
%NEWPAGE ……200
%NOCONDS ……201
%NOCREF ……202
%NOCTLS ……203
NOEMUL ……204
%NOINCL ……205
NOJUMPS ……206
%NOLIST ……207
NOLOCALS ……208
%NOMACS ……209
NOMASM51 ……210
NOMULTERRS ……211
%NOSYMS ……212
%NOTRUNC ……213
NOWARN ……214
ORG ……215
%OUT ……216
P186 ……217
P286 ……217
P286N ……217
P287 ……218
P386 ……218
P386N ……218
P387 ……219
P8086 ……219
P8087 ……219
PAGE ……220
%PAGESIZE ……221
%PCNT ……222
PNO87 ……223
%POPLCTL ……224
PROC ……225
PUBLIC ……228
PURGE ……229
%PUSHLCTL ……230
QUIRKS ……231
.RADIX ……232
RADIX ……232
RECORD ……233
REPT ……235
.SALL ……236
SEGMENT ……237
.SEQ ……240
.SFCOND ……241

SIZESTR……241
.STACK……242
STACK……242
STRUC……243
SUBSTR……245
SUBTTL……246
%SUBTTL……247
%SYMS……248
%TABSIZE……249
%TEXT……250
.TFCOND……251
TITLE……252
%TITLE……253
%TRUNC……254
UDATASEG……254
UFARDATA……255
UNION……256
USES……258
WARN……260
.XALL……261
.XCREF……262
.XLIST……262

付録A　Turbo Assemblerの構文のまとめ……263

語意に関する文法……263
valid_line……263
white_space……263
space_char……263
id_string……263
id_strng2……264
id_char……264
id_chr2……264
number_string……264
num_string……264
digits……264
digit……264
alphanums……264
alpha……264
exp……265
str_string……265
punctuation……265
MASMモードの式に関する文法……265
mexpr1……265
mexpr2……266
mexpr3……266
mexpr4……266

mexpr5……266
mexpr6……266
mexpr7……266
mexpr8……267
mexpr9……267
mexpr10……267
mexpr11……267
mexpr12……267
mexpr13……267
Idealモードの式に関する文法……268
pointer……268
type……268
pointer2……268
pointer3……268
expr……269
expr2……269
expr3……269
expr4……269
expr5……269
expr6……269
expr7……270
expr8……270
expr9……270
expr10……270

付録B　互換性の問題……271
環境変数……272
マイクロソフトのバイナリ浮動小数点フォーマット……272
Turobo AssemblerのQuirksモード……273
セグメントレジスタ間でのバイト転送……273
farラベルまたはfarプロシージャへのエラーnearジャンプ……273
=疑似命令とEQU疑似命令の型情報の消失……274
セグメントアラインメントのチェック……274
符号付き即値の算術演算命令と論理演算命令……275
MASMのバージョン5.1の機能……275

MASMバージョン5.1またはQuirksモードの機能 ……276

付録C　Turbo Assemblerの重要項目 ……277
拡張されたコマンドラインの構文 ……277
GLOBAL疑似命令 ……277
ローカルシンボル ……278
条件ジャンプの拡張機能 ……278
Idealモード ……278
UNION疑似命令/STRUCのネスティング ……278
ARGとLOCAL疑似命令 ……279
EMULとNOEMUL疑似命令 ……279
明示的なセグメントオーバライド ……279
定数セグメント ……279
386モードにおける拡張LOOP命令 ……280
拡張リスト制御機能 ……280
代替疑似命令 ……280
定義済み変数 ……280
MASMのバージョン5.0と5.1における改良点 ……281
改善されたSHLとSHRの扱い ……281

付録D　Turbo Assemblerのユーティリティ ……283
スタンドアロンMAKEユーティリティ ……284
簡単な例 ……284
メイクファイルの作成 ……286
MAKEの使い方 ……287
MAKEの動作 ……288
メイクファイルの作成 ……289
メイクファイルの構成要素 ……289
コメント ……290
明示的ルール ……290
明示ルールに関する注意 ……291
明示的ルールの例 ……292
暗黙のルール ……292
暗黙のルールに関する注意 ……295
暗黙のルールの例 ……295

コマンドリスト ……295
プレフィクス ……296
コマンド ……296
コマンドリストの例 ……297
マクロ ……298
マクロの定義 ……299
マクロの使い方 ……299
マクロに関する注意 ……299
定義済みマクロ ……300
命令 ……303
ファイルインクルード命令 ……303
条件命令 ……304
条件命令の中に使用できる式 ……305
エラー命令 ……306
定義解除命令 ……307
MAKEの使い方 ……307
コマンドラインの規則 ……307
MAKEを停止させる方法 ……308
BUILTINS.MAKファイル ……309
MAKEのメイクファイルの探し方 ……309
TOUCHユーティリティ ……309
MAKEのコマンドラインオプション ……310
MAKEのメッセージ ……310
致命的エラーメッセージ ……311
エラーメッセージ ……312
TLINK ……315
TLINKの起動 ……315
レスポンスファイルの使い方 ……317
TLINKオプション ……318
/x, /m, /sオプション ……318
/lオプション ……320
/iオプション ……320
/nオプション ……320
/cオプション ……321
/dオプション ……321

/eオプション ……321
/tオプション ……322
/vオプション ……322
/3オプション ……322
制約 ……323
メッセージ ……323
致命的エラーメッセージ ……324
非致命的エラーメッセージ ……326
警告メッセージ ……327
TLIB ……328
オブジェクトモジュールライブラリを使う利点 ……328
TLIBのコマンドラインの要素 ……329
操作リスト ……331
ファイル名とモジュール名 ……331
TLIBの操作 ……331
ライブラリの作成 ……333
レスポンスファイルの使い方 ……333
文字の大小の区別(/Cオプション) ……333
TLIBの使用例 ……334
拡張辞書の作成(/Eオプション) ……335
GREP ……336
GREPのオプション ……336
優先順位 ……337
検索文字列 ……338
正規表現の演算子 ……338
ファイルの指定 ……339
GREPの使用例 ……339
OBJXREF ……344
OBJXREFのコマンドライン ……344
コマンドラインオプション ……345
制御オプション ……345
レポートオプション ……346
レスポンスファイル ……346
自由形式のレスポンスファイル ……347
リンカのレスポンスファイル ……347

/Oオプション……347
/Nオプション……347
OBJXREFレポートの例……348
パブリックス名レポート(/RP)……349
モジュール別レポート(/RM)……350
参照レポート(/RR)(デフォルト)……350
外部参照レポート(/RX)……351
モジュールサイズレポート(/RS)……351
クラス別レポート(/RC)……352
参照されないシンボル名のレポート(/RU)……352
詳細レポート(/RV)……353
OBJXREFの使用例……353
OBJXREFのメッセージ……354
エラーメッセージ……354
警告メッセージ……354
TCREF……356
レスポンスファイル……356
TLINKとの互換性……356
スイッチ……357
出力……357
グローバル(リンカスコープ)レポート……357
ローカル(モジュールスコープ)レポート……358

付録E　エラーメッセージ……359
指示メッセージ……360
警告メッセージとエラーメッセージ……361
致命的エラーメッセージ……388

索引……391

はじめに

このマニュアルはTurbo Assemblerパッケージに添付されている2冊目のマニュアルです。1冊目のマニュアル（**ユーザーズガイド**）はすでに熟読されたことと思います。このマニュアルは最重要情報のリファレンスとしてお使いください。

この**リファレンスガイド**は，Turbo Assemblerのわかりやすい，的を射たリファレンスマニュアルです。アセンブリ言語の基礎についてもう少し復習する必要がある場合は，**ユーザーズガイド**に戻ってより詳しい説明を参照してください。

必要なハードウェアとソフトウェア

Turbo AssemblerはNEC PC-9801シリーズとIBM PCファミリ（XT, AT, PS/2）およびそれらの互換機で使うことができます。OSはMS-DOS（バージョン2.0以上），必要なメモリ容量は256Kバイトです。

Turbo Assemblerがサポートするプロセッサは8086, 80186, 80286, および80386です。数値演算コプロセッサ8087, 80287, 80387用の浮動小数点命令もサポートしています。

マニュアルについて

このマニュアルの構成は次のようになっています。

第1章：定義済みシンボル

Turbo Assemblerのイコール定義について説明します。

第2章：演算子

Turbo Assemblerのさまざまな演算子について説明します。

第3章：疑似命令

Turbo Assemblerの疑似命令の詳細をアルファベット順に説明します。

付録A：Turbo Assemblerの構文のまとめ

Turbo Assemblerの式（MASMモードとIdealモード）をバッカス・ナウア記法（BNF）で説明します。

付録B：互換性の問題

MASMモードとTurbo AssemblerのMASMモードの相違点を説明します。

付録C：Turbo Assemblerの重要項目

Turbo AssemblerがMASMに対して優れている点を説明します。

付録D：Turbo Assemblerのユーティリティ

このパッケージに添付されている6つのユーティリティMAKE, TLIB, GREP, OBJXREF, TCREFについて説明します。

付録E：Turbo Assemblerのエラーメッセージ

Turbo Assembler使用時に表示されるさまざまなエラーメッセージ（インフォメーションメッセージ，致命的エラーメッセージ，警告メッセージ，エラーメッセージなど）について説明します。

マニュアルの表記法

NEC PC-9801, IBM PC，またはそれらの互換機について言及したときは，8088, 80186, 80286, 80386（これらはすべて80x86と表記します）を使ったコンピュータを示します。MS-DOS, PC-DOSなどはすべてバージョン2.0以上のOSを示しています。

このマニュアルでは次のように書体を使いわけています。

Italics　ラベル，変数，配列を示します。文法説明ではユーザ指定項目はイタリック体で表記しています。

Boldface　疑似命令，機械命令，シンボル，演算子，コマンドラインオプションはボールド体で表記しています。

CAPITALS　機械命令，疑似命令，レジスタ，演算子はアルファベットの大文字で表記しています。

Keycaps　キーボード上のキー（***ESC, F10***など）はこの書体で表記しています。特定の機能を実行するために押すキーを表わします。

謝辞

このマニュアルでは，以下に示す製品を参照しています。

■Turbo C, Turbo Pascal, Turbo Basic, Turbo Prolog, Turbo Debuggerは，ボーランドインターナショナル社の登録商標です。
■NEC PC-9801は，日本電気株式会社の登録商標です。
■IBM PC AT/XT, PS/2, PC-DOSは，米国IBM社の登録商標です。
■8086, 8088, 80186, 80286, 80287, 80386, 80387は，米国インテル社の登録商標です。
■MS-DOSは米国マイクロソフト社の登録商標です。

製品に関するお問い合わせ

ソフトウェアおよびマニュアルに関するご意見・ご質問は，下記の**MSAカスタマサポートセンター**まで，お電話，書面またはファクシミリにてお問い合わせください。なお，お電話での受付時間は平日の9:00am～12:00amおよび1:00pm～5:00pmとなっています（土曜，日曜，祝日は休業となりますのでご了承ください)。

〒107　東京都港区南青山7-8-1　小田急南青山ビル
（株）マイクロソフトウェアアソシエイツ
カスタマサポートセンタ
Tel.03-486-1403
Fax.03-486-8905

お問い合わせの際は下記の事項をお伝えください。

■製品名とバージョン番号（Turbo Assembler 1.0など）
■製品のシリアル番号（マスタディスクに記載されているもの）
■コンピュータ名（PC-9801 VM2など）
■オペレーティングシステムとそのバージョン番号（MS-DOS 3_3など）

なお，書面にてお問い合わせの際には，CONFIG.SYSおよびAUTOEXEC.BATの内容，簡潔なプログラムリストを添付されるようお願い致します。

第1章

定義済みシンボル

Turbo Assemblerにはプログラムで使える多くの定義済みシンボルがあります。これらのシンボルの値は，ソースファイル内の使われる場所によって異なります。定義済みシンボルは，**EQU**疑似命令を使って定義されるイコール定義と似たところがあります。ソースファイル内に定義済みシンボルがあると，Turbo Assemblerは対応する現在の値をそれと置き換えます。

これらのシンボルには，テキスト（文字列）に同等なシンボル，数値に同等なシンボル，およびエリアス（alias：別名定義シンボル）があります。文字列に同等なシンボルは，文字列を指定する場所ならどこにでも使うことができます。たとえば，**DB**疑似命令を使って一連のデータバイトを初期化する場合は次のようになります。

```
NOW   DB   ??Time
```

数値の定義済みシンボルは，数値を指定する場所ならどこにでも使うことができます。次に例を示します。

```
IF ??Version GT 100h
```

エリアスは定義済みシンボルを対応する値の同義語に変換します。したがって通常のシンボル名を使う場所ならどこにでも定義済みシンボル名を使うことができます。次に例を示します。

```
ASSUME  CS:@code
```

定義済みシンボルはMASMモードでもIdealモードでも使うことができます。

アセンブル時に/**ML**コマンドラインオプションを使う場合は，定義済みシンボルはそのままの形式で使わなければなりません（つまり英大文字，小文字を区別します）。

単価記号（@）で始まる定義済みシンボルを使う場合は，次の規則が適用されます。

・シンボル名の各構成単語の先頭文字は大文字でなければなりません（ただしセグメント名エリアスは除きます）。

・それ以外の文字はすべて小文字となります。

例：

```
@FileName
@WordSize
@curseg
@fardata
```

/**ML**コマンドラインオプションを使う場合は，2つの疑問符（??）で始まる定義済みシンボルはすべて小文字でなくてはなりません。

2つの疑問符（??）で始まる定義済みシンボルを使う場合は，通常大文字と小文字を組み合わせることができます。ただし/**ML**コマンドラインオプションを使う場合は例外で，すべての文字は小文字でなくてはなりません。

@code

機能　.CODEセグメントの名前を表わすエリアスに同等なシンボルです。

説明　簡略セグメント疑似命令（**.MODEL**など）を使う場合は,**ASSUME**やセグメントオーバライドなどの式でこのシンボルを指定すれば,コードセグメントの名前を表わすことができます。

例

```
.DATA
mov  ax,@code
mov  ds,ax
ASSUME DS:@code
```

@CodeSize

機能　コードのメモリモデルを表わす数値に同等なシンボルです。

説明　**@CodeSize**は，nearコードポインタを使うスモールモデルとコンパクトモデルが指定されている場合は0に，farコードポインタを使う他のモデルを指定する場合は1に設定されます。
このシンボルを使えば，メモリモデルに従って関数へのポインタのアセンブル方法をコントロールすることができます。

例

```
IF @CodeSize EQ 0
PROCPTR   DW   PROC1      ; nearプロシージャへのポインタ
     ELSE
PROCPTR   DD   PROC1      ; farプロシージャへのポインタ
     ENDIF
```

@Cpu

機能　現在のプロセッサに関する情報を表わす数値に同等なシンボルです。

説明　**@Cpu**は，プロセッサの型を表わす値を次のように1ビットのフィールドを使って返します。

ビット	有効命令
0	8086命令
1	80186命令
2	80286命令
3	80386命令
7	特権命令（80286と80386）
8	8087数値演算プロセッサ命令
10	80287数値演算プロセッサ命令
11	80387数値演算プロセッサ命令

ここで定義されていないビットは，将来のための予備ビットです。Turbo Assembrerの将来のバージョンでも使えるように，**@Cpu**を使う際はこれらの予備ビットはマスクしておいてください。

8086プロセッサファミリは上位互換性があるため，**.286**などの疑似命令でプロセッサの型を指定するとその下位プロセッサ（8086, 80186）の命令も自動的に使用可能になります。

注意：このシンボルは，アセンブル時に.286やその関連疑似命令を使って選択したプロセッサに関する情報を返すだけです。プログラム実行時のプロセッサの型は返しません。

例

```
IPUSH = @Cpu AND 2    ; 186より上位のプロセッサでは直接プッシュする
IF IPUSH
PUSH 1234
ELSE
   mov  ax,1234
   push ax
ENDIF
```

@curseg

機能　カレントセグメントを表わすエリアスに同等なシンボルです。

説明　**@curseg**は，カレントセグメントの名前を表わします。カレントセグメントはアセンブル中に変化します。通常は簡略セグメント疑似命令（**.MODEL**など）の後の**ASSUME**やセグメントオーバライドなどの式でこのシンボルを指定すれば，カレントセグメント名を表わすことができます。

例

```
.CODE
ASSUME CS:@curseg
```

@data

機能　nearデータグループの名前を表わすエリアスに同等なシンボルです。

説明　簡略セグメント疑似命令（**.MODEL**など）を使う場合は，**ASSUME**やオーバライドなどの式でこのシンボルを使えば，すべてのnearデータセグメント（**.DATA, .CONST, .STACK**）でグループ名を共通に使うことができます。

例

```
.CODE
mov  ax,@data
mov  ds,ax
ASSUME DS:@data
```

@DataSize

機能　データのメモリモデルを表わす数値に同等なシンボルです。

説明　**@DataSize**は，nearデータポインタを使うタイニィモデル，スモールモデル，ミディアムモデルが指定されている場合は0に，farデータポインタを使うコンパクトモデルとラージモデルを指定されている場合は1に設定されます。
このシンボルを使えば，メモリモデルにしたがってデータへのポインタのアセンブル方法をコントロールすることができます。

例

```
IF @DataSize EQ 1
    lea   si,Dataptr
    mov   al,[BYTE PTR si]
ELSE
    les   si,Dataptr
    mov   al,[BYTE PTR ES:SI]
ENDIF
```

??Date

機能　今日の日付を表わす文字列に同等なシンボルです。

説明　**??Date**は今日の日付を表わす文字列に同等なシンボルです。日付の文字列の形式は，DOSの国別コードによって異なります。

参照　**??Time**

例

```
ASMDATE db    ??date
```

@fardata

機能　初期化済みのfarデータセグメントの名前を表わすエリアスに同等なシンボルです。

説明　簡略セグメント疑似命令（**.MODEL**など）を使う場合は，**ASSUME**やオーバライドなどの式でこのシンボルを指定すれば，初期化済みのfarデータセグメント（**.FARDATA**）の名前を表わすことができます。

例

```
mov  ax,@fardata
mov  ds,ax
ASSUME DS:@fardata
```

@fardata?

機能　初期化されてないfarデータセグメントの名前を表わすエリアスに同等なシンボルです。

説明　簡略セグメント疑似命令（**.MODEL**など）を使う場合は，**ASSUME**やオーバライドなどの式でこのシンボルを指定すれば，初期化されてないfarデータセグメント（**.FARDATA?**）の名前を表わすことができます。

例

```
mov  ax,@fardata?
mov  ds,ax
ASSUME ds:@fardata?
```

@FileName

機能　カレントアセンブルファイルを表わすエリアスに同等なシンボルです。

参照　??Filename

??Filename

機能　カレントアセンブルファイルを表わす文字列に同等なシンボルです。

説明　??Filanemeはアセンブル中のファイル名を表わす8文字の文字列を定義します。ファイル名が8文字に満たない場合は，スペースが埋められます。

例

```
SRCNAME DB    ??Filename    ; 常に8バイト
```

??Time

機能　現在時刻を表わす文字列に同等なシンボルです。

説明　**??Time**は現在時刻を表わすテキストに同等なシンボルです。時刻の文字列の形式は，DOSの国別コードによって異なります。

参照　**??Date**

例

```
ASMTIME DB    ??Time    ; 8 バイトの文字列
```

??Version

機能　Turbo Assemblerのバージョンを表わす数値に同等なシンボルです。

説明　上位バイトが主バージョン番号で下位バイトが副バージョン番号です。たとえばV2.1は201hで表わされます。
??Versionを使えば，Turbo Assemblerの特定バージョンの機能を利用するソースファイルを記述することができます。
MASMは**??Version**をサポートしていないため，MASMまたはTurbo Assemblerのどちらでアセンブルされるのかをプログラムで知ることができます。

例

```
IFDEF  ??Version
; TURBO ASSEMBLER STUFF
ENDIF
```

@WordSize

機能　16ビットまたは32ビットのセグメントを表わす数値に同等なシンボルです。

説明　**@WordSize**はカレントセグメントが16ビットセグメントの場合は2を，32ビットセグメントの場合は4を返します。

例

```
IF @WordSize EQ 4
   mov esp,0100h
ELSE
   mov sp,0100h
ENDIF
```

第2章

演算子

演算子を使うと複雑な式を書くことができます。式は，ソースファイルの中で，命令や疑似命令のオペランドとして，またその他のいろいろな場所に使うことができます。ソースファイルの中に式を使うと，Turbo Assemblerは，ソースファイルをアセンブルするときに式を評価し，式のかわりにその計算結果を使います。

この章では，Turbo Assemblerで使用できる演算子について詳しく説明します。

算術演算の精度

Turbo Assemblerは，.386または.386P疑似命令を使って80386プロセッサを指定したときは32ビット，指定しないときは16ビットの算術演算を使います。80386プロセッサを指定したときまたはIdealモードのときと，16ビットモードのときとで，式の計算結果が異なることがあります。たとえば

```
DW      (1000h * 1000h) / 1000h
```

は，32ビットモードでは1000h（ワード）になり，16ビットモードでは0000h（ワード）になります。16ビットモードでは乗算がオーバーフローし，結果の下位16ビットだけが保存されるからです。

演算子の優先順位

Turbo Assemblerは，次の規則に従って式を評価します。

- 優先順位の高い演算子は，優先順位の低い演算子よりも先に実行されます。
- 優先順位が同等な演算子は，式の最も左側の演算子から評価されます。
- 式中にカッコ（ ）に囲まれた部分式があるときは，その部分式が先に評価されます。カッコで囲まれた部分式は，最も高い優先順位を持ちます。

演算子の中には，IdealモードとMASMモードで優先順位の異なるものがあります。次の2つの表は，各モードにおける演算子の優先順位を示します。表の最上行は，優先順位が最も高い演算子を示します。最下行は，優先順位が最も低い演算子を示します。同じ行の演算子は，すべて同じ優先順位を持ちます。

表2.1　MASMモードの演算子の優先順位

<>, () , [] ,LENGTH, MASK, SIZE, WIDTH
.（構造体メンバ選択子）
HIGH, LOW
+, -（単項）
：（セグメントオーバーライド）
OFFSET, PTR, SEG, THIS, TYPE
*, /,MOD, SHL, SHR
+, -（二項）
EQ, GE, GT, LE, LT, NE
NOT
AND
OR, XOR
LARGE, SHORT, SMALL, .TYPE

表2.2　Idealモードの演算子の優先順位

() , [] ,LENGTH, MASK, OFFSET, SEG, SIZE, WIDTH
HIGH, LOW
+, -（単項）
*, /,MOD, SHL, SHR
+, -（二項）
EQ, GE, GT, LE, LT, NE
NOT
AND
OR, XOR
：（セグメントオーバーライド）
.（構造体メンバ選択子）
HIGH（ポインタの前）, LARGE, LOW（ポインタの前）, PTR, SHORT, SMALL, SYMTYPE

次に式中に使用できる演算子をアルファベット順に説明します。

()

機能　囲んだ式が先に評価されるようにします。

モード　MASM, Ideal

構文　(*expression*)

説明　この演算子を使うと，演算子の評価の優先順位を変更することができます。カッコで囲んだ部分式の演算子は，カッコの前および後の演算子よりも先に評価されます。

参照　**+, -, *, /, MOD, SHL, SHR**

例

```
(3 + 4) * 5      ; 35になる
 3 + 4  * 5      ; 23になる
```

*

機能　2つの整数式を掛け算します。

モード　MASM, Ideal

構文　*expression1* * *expression2*

説明　*expression1*と*expression2*は，両方とも整数の定数にならなければなりません。この演算子は，.386のアドレシングモードをサポートすることができるように，レジスタと整数を掛け算することもできます。

参照　**+, -, /, MOD, SHL, SHR**

例
```
SCREENSIZE = 25 * 80    ; 画面上の総文字数
```

＋（二項）

機能　2つの式を足し算します。

モード　MASM, Ideal

構文　*expression1* ＋ *expression2*

説明　*expression1*と*expression2*の少なくとも一方は，定数にならなければなりません。またどちらか1つの式は，アドレスでもかまいません。

参照　–, ∗,/, **MOD, SHL, SHR**

例

```
X     DW    4 DUP (?)       ;
XPTR DW     X + 4           ; バッファ内の4番目のワード
```

＋（単項）

機能　正の数であることを示します。

モード　MASM, Ideal

構文　＋ *expression*

説明　この演算子には何の効果もありません。ただ正の定数であることを明示するために使用します。

参照　–, ∗,/, **MOD, SHL, SHR**

例

```
FOO   DB   +4        ; なくても同じ+
```

ー（二項）

機能　2つの式を引き算します。

モード　MASM, Ideal

構文　*expression1* - *expression2*

説明　引き算では，次の3つのオペランドの組み合わせが使用できます。

■*expression1*と*expression2*が両方とも整定数。

■*expression1*と*expression2*が両方ともアドレス(ただし同じセグメント内のアドレスでなければならない)。
2つのアドレスを引き算すると，演算の結果は定数になります。Idealモードは，MASMモードよりもずっと厳しく*expression1*と*expression2*の整合性をチェックします。またIdealモードでは，farポインタからセグメントを引き算し，オフセットだけを残すこともできます。

■*expression1*がアドレス，*expression2*が定数。ただしその逆は許されません。演算の結果はまたアドレスになります。

参照　**+, *,/, MOD, SHL, SHR**

例

```
DATA    SEGMENT
          DW       ?
XYZ       EQU      10
VAL1      DW       XYZ      - 1      ; 定数9
VAL2      DW       ?
V1SIZE    DW       VAL2 - VAL1       ; 定数2
V1BEFORE  DW       VAL1 - 2          ; VAL1の前のDWを指す
DATA    ENDS
```

－ (単項)

機能　式の符号を変えます。

モード　MASM, Ideal

構文　- *expression*

説明　*expression*は定数にならなければなりません。*expression*が正のときは，結果は絶対値が等しい負の数になります。*expression*が負のときは，絶対値が等しい正の値になります。

参照　**+, ＊,/, MOD, SHL, SHR**

例

```
LOWTEMP DB   -10
```

.

機能　構造体のメンバを選択します。

モード　MASM, Ideal

構文　*memptr.fieldname*

説明　MASMモードでは，*memptr*はメモリアドレスを参照する任意のオペランド，*fieldname*は構造体のメンバの名前または定数式でもかまいません。*memptr*が構造体の名前のときも，*fieldname*はその構造体のメンバである必要はありません。この演算子の働きは，＋演算子と非常によく似ています。すなわち構造体内の*fieldname*のオフセットを，構造体*memptr*のメモリアドレスに加算します。ただし.演算子は，そのフィールド名のサイズも返します。

Idealモードでは，この演算子の働きはもっと厳密に定義されています。*memptr*は構造体を指すポインタ，*fieldname*はその構造体のメンバでなければなりません。したがって異なる構造体に同じフィールド名を使うことができます。*memptr*のかわりにベースまたはインデックスレジスタを使いたいときは，レジスタ名の前に，アクセスしたい構造体の名前に対する型キャストをつけなければなりません（下の例を参照してください)。

参照　**STRUC**

例

```
X STRUC
MEMBER1 DB      ?
MEMBER2 DW      ?
X ENDS
XINST   X       <>

                ; X型の構造体変数XINST
; MASMモード
        mov    [bx].Member2,1
; Idealモード
        mov   [(X PTR bx).Member2],1    ; 型キャストに注意
```

/

機能　2つの整数式を割り算します。

モード　MASM, Ideal

構文　*expression1* / *expression2*

説明　*expression1*と*expression2*は，両方とも整数にならなければなりません。演算の結果は，*expression1*が*expression2*で割り算されます。余りは捨てられます。**MOD**演算子を使い，/演算子に指定したのと同じオペランドを指定すると，余りを求めることができます。

参照　**+, −, *, MOD, SHL, SHR**

例

```
x = 55 / 10       ; = 5 (整数の割り算)
```

:

機能 セグメントまたはグループをオーバーライドします。

モード MASM, Ideal

構文 *segorgroup* : *expression*

説明 この演算子は，*expression*が参照するアドレスを，強制的に，指定されたセグメントまたはグループの先頭からの相対アドレスにします。この演算子を使うと，デフォルトの方法と異なる方法で*expression*にアクセスすることができます。

*segorgroup*は，次の方法で指定できます。

- セグメントレジスタ CS, DS, ES,またはSS（**P386**または**P386N**疑似命令で80386プロセッサを指定している場合には，さらにFSまたはGS）
- **SEGMENT**疑似命令で定義されたセグメント名
- **GROUP**疑似命令で定義されたグループ名
- **SEG**演算子で始まる*expression*

例

```
        mov cl,es:[si+4]
PTR      DD    DGROUP:MEMVAR
```

?

機能　確保されたデータ領域を初期化しません。

モード　MASM, Ideal

構文　*Dx* ?

説明　*Dx*は，データ定義疑似命令（**DB, DD**など）の1つを表わします。データを定義したいが，確保した領域にある特定の初期値を入れたくないときは，この演算子を使うことができます。

データを使用する前にそのデータ領域をプログラムにより初期化するときは，**?**を使うべきです。0でなく**?**を使うことにより，ソースコードの中の初期化の手順が明確になります。

DUP式の中または**DUP**式の外部にただ1つの値として を指定したときは，オブジェクトコードは生成されません。初期化される値を含む**DUP**式の中に**?**を使うと，0として扱われます。

参照　**DUP**

例

```
MyBuf DB 20 DUP (?)     ; 20バイト確保し，初期化はしない
```

[] 演算子

機能　間接メモリオペランドまたは加算を指定します。

モード　MASM, Ideal

構文　[*expression1*] [*expression2*]

説明　この演算子の働きは，*MASM*モードと*Ideal*モードで非常に異なります。

*MASM*モードでは，加算演算子としての働きもします。すなわち*expression1*を*expression2*に足し算します。オペランドの組み合わせについては同じ制約があてはまります。たとえば*expression1*と*expression2*は，両方ともアドレスになることはできません。[] はまた，BX, BP, SI, DIレジスタを用いたレジスタ間接メモリオペランドを示すこともできます。間接レジスタは [] 内に囲まなければなりません。間接オフセットは [] の中に入れても，入れなくてもかまいません。

Idealモードでは，[]はいつでも"メモリ参照"を意味します。メモリアドレスの内容を指すオペランドは，カッコの中に囲まなければなりません。そのためオペランドが即値であるか，メモリ参照であるか，オフセットであるかが明確になります。

参照　+

例

```
; MASMモード
    mov al,BYTE PTR ES:[BX]
    mov al,cs:10h
; Idealモード
    mov al,[BYTE PTR ES:BX]
    mov al,[cs:10h]
```

AND

機能　ビットごとの**AND**

モード　MASM, Ideal

構文　*expression1* **AND** *expression2*

説明　この演算子は，*expression1*と*expression2*の各ビットを，ビットごとに**AND**演算します。演算の結果は，両方の式のビットが1のビット位置が1，その他のビット位置が0になります。

参照　**NOT, OR, XOR**

例

```
mov   al,11110000b AND 10100000B   ; ALに10100000Bをロード
```

BYTE

機能　式を強制的にバイトサイズにします。

モード　Ideal

構文　**BYTE** *expression*

説明　*expression*はアドレスでなければなりません。結果は，式の元のサイズにかかわらず，同じメモリアドレスを指す**BYTE**サイズの式になります。

この演算子は，通常，前方参照される式のサイズを指定するため，またはそれ自体からはサイズを決めることができないレジスタ間接式のサイズを指定するために使います。

MASMモードでこれと同じことを行なうには，**PTR**演算子を使い，その前に**BYTE**型を指定しなければなりません。

参照　**PTR**

例

```
mov   BYTE [bx],1      ; バイト即値転送
mov   BYTE X,1         ; 前方参照
X     DB   0
```

DUP

機能　データ領域の確保を繰り返します。

モード　MASM, Ideal

構文　*count* **DUP**　(*expression* [,*expression*] ...)

説明　**DUP**演算子は，データ定義疑似命令（**DB, DW**など）の後に使用されます。*count*は繰り返しの回数，式*expression*は確保するデータ領域に入れる初期値です。各式の値は，**DUP**の前のデータ定義疑似命令により指定されるデータのサイズに適合していなければなりません。

式の中にまた**DUP**演算子を入れることができます。17レベルまでネストできます。

式はいつでもカッコ（　）で囲まなければなりません。

例

```
WRDBUF  DW  40 DUP (1)          ; 40ワードを確保して1に初期化する
SQUARE  DB   4 DUP (4 DUP (0))  ; 4×4の配列を確保して0に初期化する
```

DWORD

機能　式を強制的にダブルワードサイズにします。

モード　Ideal

構文　**DWORD** *expression*

説明　*expression*はアドレスでなければなりません。結果は，式の元のサイズにかかわらず，同じメモリアドレスを指す**DWORD**サイズの式になります。
この演算子は，通常，前方参照される式のサイズを指定するため，またはそれ自体からはサイズを決めることができないレジスタ間接式のサイズを指定するために使います。
MASMモードでこれと同じことを行なうには，**PTR**演算子を使い，その前に**DWORD**型を指定しなければなりません。

参照　**PTR**

例

```
call  dword fptr     ; 前方参照
                     ; 間接呼び出し
fptr  DD    MYFUNC
```

EQ

機能　2つの式が等しいときに真を返します。

モード　MASM, Ideal

構文　*expression1* **EQ** *expression2*

説明　*expression1*と*expression2*は，両方とも定数にならなければなりません。**EQ**は，2つの式の結果が等しいときに真（-1）を，異なるときに偽（0）を返します。

EQは，*expression1*と*expression2*を，最上位のビットを符号ビットとする32ビットの符号つき数値と見なします。これは-1 **EQ** 0FFFFFFFFhが真になることを意味します。

参照　**NE, LT, LE, GT, GE**

例

```
ALIE    = 4 EQ 3    ; = 0  (偽)
ATRUTH = 6 EQ 6    ; = -1 (真)
```

FAR

機能　式を強制的にfarコードポインタにします。

モード　Ideal

構文　**FAR** *expression*

説明　*expression*はアドレスでなければなりません。結果は，式の元の型にかかわらず，同じメモリアドレスを指し，セグメントとオフセットの両方を持つfarポインタになります。

この演算子は通常，ソースファイルのより後のほうでfar宣言されている前方参照されるラベルに対して，呼び出しを行なったりジャンプしたりするために使用します。

MASMモードで同じことをするためには，**PTR**演算子を使い，その前に**FAR**型の指定をしなければなりません。

参照　**NEAR**

例

```
      call FAR ABC      ; 前方参照
ABC   PROC FAR
```

FWORD

機能　式を強制的に32ビットのfarポインタサイズにします。

モード　Ideal

構文　**FWORD** *expression*

説明　*expression*はアドレスでなければなりません。結果は，同じメモリアドレスを指すFWORDサイズの式になります。

この演算子は，通常，前方参照される式のサイズを指定したり，それ自体からはサイズを決めることができないレジスタ間接式のサイズを明示的に指定するために使用します。

MASMモードで同じことをするためには，**PTR**演算子を使い，その前に**FWORD**型の指定をしなければなりません。

参照　**PTR, PWORD**

例

```
.386
call fword [bx]          ; far間接48ビットコール
jmp fword funcp          ; 前方参照
funcp   DF  myproc       ; PROCを指す間接ポインタ
```

GE

機能　式がもう1つの式より大きいかまたは等しいとき真を返します。

モード　MASM, Ideal

構文　*expression1* **GE** *expression2*

説明　*expression1*と*expression2*は，定数にならなければなりません。**GE**は，*expression1*が*expression2*よりも大きいかまたは等しいときに真(-1)を返します。*expression1*が*expression2*よりも小さいときは偽（0）を返します。

GEは，*expression1*と*expression2*を，最上位のビットを符号ビットとする33ビットの符号つき数値とみなします。これは1 **GE** -1が真になること，1 **GE** 0FFFFFFFFhが偽になることを意味します。

参照　**EQ, GT, LE, LT, NE**

例

```
TRUTH =  5 GE 5
AFIB  =  5 GE 6
```

GT

機能　式がもう1つの式より大きいとき真を返します。

モード　MASM, Ideal

構文　*expression1* **GT** *expression2*

説明　*expression1*と*expression2*は,定数にならなければなりません。**GT**は，*expression1*が*expression2*よりも大きいときに真(-1)を返します。*expression1*が*expression2*よりも小さいかまたは等しいときは偽（0）を返します。

GTは，*expression1*と*expression2*を，最上位のビットを符号ビットとする33ビットの符号つき数値とみなします。これは1 **GT** -1が真になること,1 **GT** 0FFFFFFFFhが偽になることを意味します。

参照　**EQ, GE, LE, LT, NE**

例

```
AFACT =  10 GT 9
NOTSO =  10 GT 11
```

HIGH

機能　式の上位部を返します。

モード　MASM, Ideal

構文　**HIGH** *expression*

Idealモードのみ：

type **HIGH** *expression*

説明　**HIGH**は，*expression*の上位8ビットを返します。*expression*は定数にならなければなりません。

Idealモードでは，**HIGH**は，**LOW**との組み合わせにより，データから任意のフィールドを取り出すための強力な手段になります。*type*は，*expression*から取り出すフィールドのサイズを指定します。通常のサイズ指定子(**BYTE, WORD, DWORD**など)のどれでもかまいません。1つの*expression*に複数の**HIGH**または**LOW**を適用することもできます。たとえば

```
BYTE LOW WORD HIGH DBLVAL
```

は，ダブルワードDBLVALの下から3番目のバイトを指すバイトアドレスになります。

参照　**LOW**

例

```
; MASMおよびIdealモード
Magic   EQU   1234h
        mov   cl, HIGH Magic
        Ideal
; Idealモードのみ
Big     DD    12345678h
        mov   ax,[word HIGH Big]  ; 1234hをAXにロード
```

LARGE

機能　式のオフセットサイズを32ビットにします。

モード　MASM, Ideal

構文　**LARGE** *expression*

説明　*expression*は，どのような式でもかまいません。**LARGE**はそれを32ビットのオフセットに変換します。

この演算子は，通常，演算のサイズについての曖昧さを取り除くために使用します。たとえば**P386**疑似命令により80386プロセッサを指定したとき，次のコードは，セグメントと16ビットのオフセットからなるfarコール，または32ビットのオフセットを使ったnearコールのどちらにも解釈できます。

```
jmp [DWORD PTR ABC]
```

LARGE演算子を使うと，この曖昧さを取り除くことができます。

```
jmp LARGE [DWORD PTR ABC]    ; 32ビットオフセットのnearコール
```

この例では，**LARGE**演算子はカッコ[]の外にあります。そのためメモリから読み出された**DWORD**の解釈を指定します。**LARGE**がカッコの中にあるときは，メモリから読み出されたオペランドのサイズではなく，オペランドを読み出すアドレスのサイズを決定します。たとえば次のコード

```
jmp  LARGE [LARGE DWORD PTR XYZ]
```

は，*XYZ*を32ビットのオフセットとして扱い，そのアドレスからやはり32ビットのオフセットであるターゲットアドレスを読み出して，間接ジャンプします。

また**LARGE**を使うと，前方参照される変数にアクセスするときに，間違った仮定を避けることができます。

```
mov  ax,[LARGE FOOBAR]  ;  FOOBARは，USE32セグメント内にある
```

参照　**SMALL**

LE

機能　式がもう1つの式より小さいかまたは等しいときに真を返します。

モード　MASM, Ideal

構文　*expression1* **LE** *expression2*

説明　*expression1*と*expression2*は両方とも定数にならなければなりません。**LE**は, *expression1*が*expression2*より小さいかまたは等しいときに真(-1)を返します。*expression1*が*expression2*よりも大きいときは偽（0）を返します。

LEは, *expression1*と*expression2*を, 最上位のビットを符号ビットとする33ビットの符号つき数値とみなします。これは1 **LE** -1が偽になること, 1 **LE** 0FFFFFFFFhが真になることを意味します。

参照　**EQ, GE, GT, LT, NE**

例

```
YUP = 5 LE 6    ; 真 = -1
```

LENGTH

機能　DUPにより繰り返し確保された要素の繰り返し回数を返します。

モード　MASM, Ideal

構文　**LENGTH** *name*

説明　*name*は，データ定義疑似命令（**DB, DD**など）で定義されたデータを表わすシンボルです。**LENGTH**は，*name*の中の繰り返し確保された要素の繰り返し回数を返します。

この演算子は，*name*が**DUP**演算子を使って宣言されていないときは，いつでも1を返します。

また*name*が複数の項からなる配列(値をカンマで区切って指定)のときも，1を返します。

参照　**SIZE, TYPE**

例

```
MSG      DB       "Hello"
array    DW       10 DUP (0)
numbrs   DD       1,2,3,4
var      DQ       ?
lmsg     = LENGTH MSG        ; =  1, DUPなし
larray   = LENGTH array      ; = 10, DUPの繰り返し回数
lnumbrs  = LENGTH numbrs     ; =  1, DUPなし
lvar     = LENGTH var        ; =  1, DUPなし
```

LOW

機能　式の下位部を返します。

モード　MASM, Ideal

構文　**LOW** *expression*

Idealモードのみ：

type **LOW** *expression*

説明　*expression*は定数にならなければなりません。**LOW**は式の下位8ビットを返します。

Idealモードでは，**LOW**は，**HIGH**との組み合わせにより，データ項から任意のフィールドを取り出すための強力な手段になります。*type*は*expression*から取り出すフィールドのサイズを指定します。通常のサイズ指定子(**BYTE, WORD, DWORD**など)のどれでもかまいません。1つの*expression*に複数の**HIGH**または**LOW**を適用することもできます。たとえば

```
BYTE LOW WORD HIGH DBLVAL
```

は，ダブルワード**DBLVAL**の下から3番目のバイトを指すバイトアドレスになります。

参照　**HIGH**

例

```
; MASMおよびIdealモード
Magic   equ   1234h
        mov   bl, LOW Magic
        Ideal
; Idealモードのみ
Big     DD    12345678h
        mov   ax,[word LOW Big]    ; 5678hをAXにロード
```

LT

機能　式がもう1つの式よりも小さいときに真を返します。

モード　MASM, Ideal

構文　*expression1* **LT** *expression2*

説明　*expression1*と*expression2*はどちらも定数にならなければなりません。**LT**は, *expression1*が*expression2*より小さいときに真(-1)を返します。*expression1*が*expression2*よりも大きいかまたは等しいときは偽（0）を返します。

LTは，*expression1*と*expression2*を，最上位のビットを符号ビットとする33ビットの符号つき数値とみなします。これは1 **LT** -1が偽になること, 1 **LT** 0FFFFFFFFhが真になることを意味します。

参照　**EQ, GE, GT, LE, NE**

例

```
JA = 3 LT 4    ; 真= -1
```

MASK

機能　レコードのフィールドに対するビットマスクを返します。

モード　MASM, Ideal

構文　**MASK** *recordfieldname*

MASK *record*

説明　*recordfieldname*は，すでに定義されているレコードのフィールドの名前です。**MASK**は，そのレコードの中の*recordfieldname*の位置に対応するビットをオンにした値を返します。

MASKにレコード名を指定したときは，そのレコードのすべてのフィールドに対応するビットをオンにした値を返します。

マスク値をレコードの全体と**AND**演算することにより，レコードの個々のフィールドを分離することができます。

参照　**WIDTH**

例

```
stat    record a:3,b:4,c:5
newstat stat <0,2,1>

mov   al,newstat          ; レコードを取得
and   al,MASK b           ; Bを分離
mov   al,MASK stat        ; レコード全体に対するマスクを取得
```

MOD

機能　2つの式の割り算の余りを返します。

モード　MASM, Ideal

構文　*expression1* **MOD** *expression2*

説明　*expression1*と*expression2*は両方とも整定数にならなければなりません。演算の結果は，*expression1*を*expression2*で割った余りになります。

参照　**+, -, *, /, SHL, SHR**

例

```
REMAINS = 17 MOD 5    ; = 2
```

NE

機能　2つの式が等しくないときに真を返します。

モード　MASM, Ideal

構文　*expression1* **NE** *expression2*

説明　*expression1*と*expression2*は，両方とも定数にならなければなりません。**NE**は，2つの式が等しくないときに真（-1）を返します。2つの式が等しいときは偽（0）を返します。

NEは，*expression1*と*expression2*を，最上位のビットを符号ビットとする32ビットの符号つき数値とみなします。これは-1 **NE** 0FFFFFFFFhが偽になることを意味します。

参照　**EQ, GE, GT, LE, LT**

例

```
aint = 10 NE 10    ; 偽= 0
```

NEAR

機能　式を強制的にnearコードポインタにします。

モード　Ideal

構文　**NEAR** *expression*

説明　*expression*はアドレスでなければなりません。結果は，式の元の型にかかわらず，同じアドレスを指すオフセットだけの**NEAR**ポインタになります。
この演算子を使うと，nearジャンプ命令やnearコール命令を使って，farラベルやfarプロシージャに対して呼び出しを行なったり，分岐したりすることができます。代表的な使い方は，下の例を参照してください。

参照　**FAR**

例

```
        Ideal
proc farp far
; プロシージャの本体
        endp farp

; まだ同じセグメント内
        push cs
        call NEAR farp       ; farコールよりもコードサイズが小さくて高速
```

NOT

機能　ビットごとに補数をとります。

モード　MASM, Ideal

構文　**NOT** *expresion*

説明　**NOT**は，式のすべてのビットを反転します。0のビットは1に，1のビットは0になります。

参照　**AND, OR, XOR**

例

```
mov al,NOT 11110011b       ; 00001100bをalにロード
```

OFFSET

機能　セグメント内のオフセットを返します。

モード　MASM, Ideal

構文　**OFFSET** *expression*

説明　*expression*は，メモリアドレスを参照するどのような式でもかまいません。**OFFSET**は,セグメントの先頭と式が参照するアドレスの間のバイト数を表わす整数を返します。

簡略セグメント疑似命令（**MODEL**など）を使っているときは，**OFFSET**は，自動的に，カレントセグメントが所属するグループの先頭からのオフセットを返します。ノーマルセグメンテーション疑似命令を使用していて，セグメントでなくグループの先頭からのオフセットを得たいときは，*expression*の一部としてそのグループを指定しなければなりません。たとえば

```
mov   si,OFFSET BUFFER
```

は，**BUFFER**を含むセグメントが**DGROUP**の最初のセグメントであるとき以外は，

```
mov   si,OFFSET DGROUP:BUFFER
```

と同じではありません。

参照　**SEG**

例

```
.DATA
msg    DB "Starting analysis"
.CODE
mov    si,OFFSET MSG              ; MSGのアドレス
```

OR

機能　ビットごとのOR

モード　MASM, Ideal

構文　*expression1* **OR** *expression2*

説明　**OR**は，*expression1*と*expression2*の各ビットを，ビットごとに**OR**演算します。演算結果は，どちらか一方または両方の式のビットが1のビット位置が1，それ以外のビット位置が0になります。

参照　**AND, NOT, XOR**

例

```
mov   al,11110000b OR 10101010b   ; 11111010bをALにロード
```

PROC

機能　式を強制的にnearコードポインタまたはfarコードポインタにします。

モード　Ideal

構文　**PROC** *expression*

説明　*expression*はアドレスでなければなりません。結果は，使用するメモリモデルに応じて，nearポインタまたはfarポインタになります。**.MODEL**疑似命令を使って**TINY**, **SMALL**,または**COMPACT**メモリモデルを指定しているときは，nearポインタになります。それ以外のときはfarポインタになります。

PROCは，通常，簡略セグメント疑似命令を使っているときに，前方参照されるプロシージャを呼び出すために使用します。下の例は代表的な使い方を示します。MASMモードでこれと同じことを行なうには，**PTR**演算子を使い，その前に**PROC**型を指定しなければなりません。

参照　**NEAR, FAR**

例

```
     .MODEL LARGE
     CODE
     IDEAL
call PROC Test1

     PROC Test1
```

PTR

機能　式を強制的に指定したサイズにします。

モード　MASM, Ideal

構文　*type* **PTR** *expression*

説明　*type*は次のどれかでなければなりません。

■データに対しては, **BYTE, WORD, DWORD, FWORD, PWORD, QWORD, TBYTE, DATAPTR**

■コードに対しては, **PROC, NEAR, FAR**

*expression*は定数にならなければなりません。結果は, *expression*の元のサイズにかかわらず, 同じメモリアドレスを指すtypeサイズの式になります。

この演算子は, 通常, 前方参照される式のサイズを定義したり, それ自体からはサイズがわからないレジスタ間接式のサイズを指定するために使用します。

Idealモードでは**PTR**演算子は不要です。*expression*の前に直接*type*を指定するだけで済みます。

参照　**BYTE, WORD, DWORD, FWORD, PWORD, QWORD, TBYTE, DATAPTR, NEAR, FAR, PROC**

例

```
mov  BYTE PTR[SI],10     ; バイト即値モード
fld  qword PTR val       ; valをクワッドワード型の浮動小数点数に変換
val  DQ  1234.5678
```

PWORD

機能　式を強制的に32ビットのfarポインタサイズにします。

モード　MASM, Ideal

構文　**PWORD** *expression*

説明　*expression*はアドレスでなければなりません。結果は，式の元のサイズにかかわらず，同じメモリアドレスを指す**PWORD**サイズの式になります。

参照　**FWORD**

QWORD

機能　式を強制的に8バイトサイズにします。

モード　Ideal

構文　**QWORD** *expression*

説明　*expression*はアドレスを参照しなければなりません。結果は，式の元のサイズにかかわらず，同じメモリアドレスを指す**QWORD**サイズの式になります。

この演算子は，通常，前方参照される式のサイズを指定するため，またはそれ自体からはサイズがわからないレジスタ間接式のサイズを指定するために使います。MASMモードでこれと同じことを行なうには，**PTR**演算子を使い，その前に**QWORD**型を指定しなければなりません。

参照　**PTR**

例

```
fadd   QWORD [BX]       ; サイズなし間接
fsubp  QWORD X          ; 前方参照
X      DQ  1.234
```

SEG

機能　式のセグメントアドレスを返します。

モード　MASM, Ideal

構文　**SEG** *expression*

説明　*expression*は，メモリアドレスを参照するどのような式でもかまいません。**SEG**は，参照されるメモリアドレスのセグメント部を表わす定数を返します。

参照　**OFFSET**

例

```
.DATA
temp   DW  0
.CODE
mov   ax,SEG temp
mov   ds,ax              ; セグメントレジスタをセットアップする
ASSUME DS:SEG TEMP       ; DSがSEG TEMPであることをアセンブラに知らせる
```

SHL

機能　式の値を左にシフトします。

モード　MASM, Ideal

構文　*expression* **SHL** *count*

説明　*expression*は定数，*count*は整数にならなければなりません。**SHL**は，*expression*のビットを*count*回左にシフトします。右からシフトインされるビットの値は0です。左からシフトアウトされるビットは失われます。

参照　**SHR**

例

```
mov  al,00000011b SHL 3   ; 00011000Bをalにロード
```

SHORT

機能　式を強制的にショートコードポインタにします。

モード　MASM, Ideal

構文　**SHORT** *expression*

説明　*expression*は，カレントコードセグメント内のアドレスを参照します。**SHORT**は，*expression*がカレントコードアドレスから±128の範囲内であることをアセンブラに知らせます。それによりアセンブラは，ショートジャンプ命令を生成します。**SHORT**を使う必要があるのは，前方参照**JMP**命令だけです。なぜならアセンブラは，*expression*の距離が±128の範囲内であることがわかっているときは，自動的にショートジャンプを生成するからです。

参照　**NEAR, FAR**

例

```
        jmp SHORT Done    ; ショートジャンプ命令を生成
        .                 ;
        .                 ; この間のコードは128バイト以下
Done:
```

SHR

機能　式の値を右にシフトします。

モード　MASM, Ideal

構文　*expression* **SHR** *count*

説明　*expression*は定数，*count*は整数にならなければなりません。**SHR**は，*expression*のビットを*count*回右にシフトします。左からシフトインされるビットの値は0です。右からシフトアウトされるビットは失われます。

参照　**SHL**

例

```
mov  al,80h SHR 2   ; 20hをalにロード
```

SIZE

機能 定義されたデータのサイズを返します。

モード MASM, Ideal

構文 **SIZE** *name*

説明 *name*は，データ定義疑似命令（**DB, DD**など）で定義されたデータを参照するシンボルです。

MASMモードでは，**SIZE**は，**LENGTH**値と**TYPE**値の積を返します。したがってこの演算子は，*name*に複数の項があっても，またDUP演算子がネストされていても，それを考慮に入れません。

Idealモードでは，**SIZE**は，データが占めているバイト数を返します。そのデータが配列のように複数の項目を含むときは，データ定義疑似命令の後の最初の項目が占めるバイト数を返します。

参照 **LNEGTH, TYPE**

例

```
    msg      DB      'Hello'
    array    DW      10 DUP (4 DUP (1), 0)
    numbrs   DD      1,2,3,4,5
    var      DQ      ?
; MASMモード
    smsg    = SIZE msg       ;  1 (LENGTH = 1)  × (TYPE =1)
    sarray  = SIZE array     ; 20 (LENGTH = 10) × (TYPE =2)
    snumbrs = SIZE numbrs    ;  4 (LENGTH = 1)  × (TYPE =4)
    svar    = SIZE var       ;  8 (LENGTH = 1)  × (TYPE =8)
; Idealモード
    smsg    = SIZE msg       ; 5
    sarray  = SIZE array     ; 2
    snumbrs = SIZE numbrs    ; 4
    svar    = SIZE var       ; 8
```

SMALL

機能　式のオフセットのサイズを16ビットにします。

モード　MASM, Ideal

構文　**SMALL** *expression*

説明　*expression*は，どのような式でもかまいません。**SMALL**は，それを16ビットのオフセットに変換します。この演算子は，通常，演算のサイズの曖昧さを取り除くために使用します。たとえば**P386**疑似命令で80386プロセッサを指定しているとき，

```
jmp  [DWORD PTR ABC]
```

は，セグメントと16ビットのオフセットからなるfarコール，または32ビットのオフセットを使ったnearコールのどちらにも解釈できます。**SMALL**演算子を使ってこれを

```
jmp  SMALL [DWORD PTR ABC]      ; 16ビットオフセットのfarコール
```

と指定すると，曖昧さを取り除くことができます。この例では**SMALL**はカッコ[]の外にあり，読み出された**DWORD**の解釈を指定します。**SMALL**がカッコ内にあるときは，読み出されたオペランドのサイズではなく，オペランドを読み出すアドレスのサイズを指定します。たとえば

```
CODE  SEGMENT USE32
jmp   SMALL [SMALL DWORD PTR XYZ]
```

と指定すると，アセンブラは，*XYZ*を16ビットのオフセットとセグメントからなる4バイトのポインタとして扱います。そしてアドレス*XYZ*から読み出したアドレスを，同じように16ビットのオフセットとセグメントからなる4バイトのポインタとして扱い，そのアドレスに間接ジャンプします。

参照　**LARGE**

SYMTYPE

機能　シンボルを記述するバイト（1バイト）を返します。

モード　Ideal

参照　.TYPE

TBYTE

機能　式を10バイトのサイズにします。

モード　Ideal

構文　**TBYTE** *expression*

説明　*expression*はアドレスでなければなりません。結果は，*expression*の元のサイズにかかわらず，同じメモリアドレスを指す**TBYTE**サイズの式になります。

TBYTEは，通常，前方参照される式のサイズを定義したり，それ自体ではサイズがわからないレジスタ間接式のサイズを指定するために使用します。

MASMモードでこれと同じことを行なうには，**PTR**演算子を使い，その前に**TBYTE**型を指定しなければなりません。

参照　**PTR**

例

```
fld   TBYTE [BX]    ; サイズのない間接式
fst   TBYTE X       ; 前方参照
X     DT     0
```

THIS

機能　カレントセグメントとロケーションカウンタが示すアドレスにオペランドを生成します。

モード　MASM, Ideal

構文　**THIS** *type*

説明　*type*は，オペランドのサイズと，そのオペランドがコードまたはデータのどちらを参照するかを指定します。

*type*は次のどれかでなければなりません。

- **NEAR, FAR, PROC**（**PROC**は，**MODEL**疑似命令で指定されたメモリモデルにより，**NEAR**または**FAR**のどちらかと同じになります)。
- **BYTE, WORD, DWORD, FWORD, PWORD, QWORD, TBYTE, DATAPTR**または構造体名

この演算子は，通常，**EQU**疑似命令および=疑似命令で変数を定義するために使います。

例

```
ptr1  EQU THIS WORD   ; 次の文と同じ
ptr2  LABEL WORD
```

.TYPE

機能　シンボルを記述するバイト（1バイト）を返します。

モード　MASM

構文　**.TYPE** *name*

説明　*name*はシンボルで，ソースファイルの中で定義されていても，定義されていなくてもかまいません。**.TYPE**は，シンボルを記述するバイトを返します。このバイトの各フィールドの意味は次のとおりです。

ビット	説明
0	プログラムシンボル
1	データシンボル
2	定数
3	直接アドレシングモード
4	レジスタ
5	定義されているシンボル
7	外部シンボル

ビット2と3が両方とも1のときは，その式には［BX］のようなレジスタ間接が使われています。

.TYPEが0を返したときは，その式は定義されていないシンボルを含みます。

.TYPEは，通常，種類の異なる引数の処理の仕方を決めるためにマクロの中で使用します。

参照　**SYMTYPE**

例

```
IF (.TYPE ABC) AND 3             ; ABCはセグメントに関係するか？
    ASSUME  ds:SEG ABC
    mov     ax,SEG ABC
    mov     ds,ax
ENDIF
```

TYPE

機能　シンボルのサイズまたは型を示す数を返します。

モード　MASM, Ideal

構文　**TYPE** *expression*

説明　**TYPE**は，式の型に基づいて次の値を返します。

BYTE	**1**
WORD	**2**
DWORD	**4**
FWORD	**6**
PWORD	**6**
QWORD	**8**
TBYTE	**10**
NEAR	**0FFFFh**
FAR	**0FFFEh**
定数	**0**
構造体	**構造体内のバイト数**

参照　**LENGTH, SIZE**

例

```
bvar     DB  1
darray   DD  10 DUP (1)
X        STRUC
         DW       ?
         DT       ?
X        ENDS
fp       EQU      THIS FAR
tbvar = TYPE bvar                   ; = 1
tdarray = TYPE darray               ; = 4
tx = TYPE x                         ; = 12
tfp = TYPE fp                       ; 0FFFEh
```

UNKNOWN

機能　式から型情報を取り除きます。

モード　Ideal

構文　**UNKNOWN** *expression*

説明　*expression*はアドレスです。結果は，型情報（**BYTE, WORD**など）が取り除かれている以外は同じ式になります。

あるメモリ変数から**UNKNOWN**演算子で型情報を取り除くと，その変数に異なるデータ型でアクセスすることができます。ただしアクセスするときにデータ型を指定しないとエラーになります。この機能は，ある場所を共用体のような使い方して，複数の異なるデータ型を格納するときに便利です。ある場所を参照するメモリ変数に，型情報を取り除いて別の名前をイコール定義すると，その場所に異なるデータ型を格納したり読み出したりすることができます（下の例を参照してください）。

また**UNKNOWN**サイズを指定したアドレスは，1つのオペランドにレジスタ間接メモリ参照を使い，もう1つのオペランドにもレジスタを使用してそのレジスタにより演算のサイズを指定するのと同じような使い方をすることができます。すなわちある名前を**UNKNOWN**と定義すると，その名前を[BX]のようなサイズ不定のレジスタ式と同じように扱うことができます。

MASMモードでこれと同じことを行なうには，**PTR**演算子を使い，その前に**BYTE**型を指定しなければなりません。

参照　**PTR**

例

```
    .DATA
workbuf DT  0                   ; ワークバッファを定義して初期値を0にする
workptr EQU UNKNOWN WORKBUF     ; サイズの不定なポインタにする
    .CODE

; 例1
    mov  [BYTE PTR WORKPTR],1   ; 1バイトを格納
    fstp [QWORD PTR WORKPTR]    ; クワッドバイト(8バイト)を格納
    mov  [WORKPTR],1            ; エラー, 型指定がない

; 例2
    mov  al,[WORKPTR]           ; 問題なし
    mov  ax,[WORKPTR]           ; これも問題なし
```

WIDTH

機能　レコードのフィールドのビット幅を返します。

モード　MASM, Ideal

構文　**WIDTH** *recordfieldname*
WIDTH *record*

説明　*recordfieldname*は，それ以前に定義されているレコードのフィールドの名前です。**WIDTH**は,そのレコード内でフィールド*recordfieldname*が占めるビット数を返します。

WIDTHの後にレコード名を指定したときは,そのレコードの総ビット数を返します。

参照　**MASK**

例

```
; MACRO DETERMINES MAXIMUM VALUE FOR A FIELD
maxval   MACRO FIELDNAME
value=2
    REPT WIDTH FIELDNAME - 1
value = value * 2
    ENDM
value = value - 1
    ENDM
```

WORD

機能　式を強制的にワードサイズにします。

モード　Ideal

構文　**WORD** *expression*

説明　*expression*はアドレスでなければなりません。結果は，式の元のサイズにかかわらず，同じメモリアドレスを指す**WORD**サイズの式になります。

WORDは，通常，前方参照される式のサイズを定義するため，またはレジスタ間接式のサイズを指定するために使います。

MASMモードで同じことをするためには，**PTR**演算子を使い，その前に**WORD**型の指定をしなければなりません。

参照　**PTR**

例

```
mov   WORD [BX],1        ; ワード即値転送
mov   WORD X,1           ; 前方参照
X     DW   0
```

XOR

機能　ビットごとのXOR演算をします。

モード　MASM, Ideal

構文　expression1 **XOR** *expression2*

説明　**XOR**は，*expression1*と*expression2*の各ビットを，ビットごとに排他**OR**演算します。演算の結果は，どちらか一方の式のビットが1で他方の式のビットが0のビット位置が1，それ以外のビット位置が0になります。

参照　**AND, NOT, OR**

例
```
mov  al,11110000b XOR 11000011b    ; AL = 00110011b
```

特別のマクロ演算子

マクロを呼び出すとき，およびマクロの定義や繰り返しブロックの定義の内部では，次の特別のマクロ演算子を使うことができます。またこのマクロ演算子に引数を指定して，条件アセンブリ疑似命令の条件を設定することができます。

特別のマクロ演算子とその働きは次のとおりです。

&	代入演算子
< >	リテラル文字列演算子
!	引用文字演算子
%	式評価演算子
;;	コメント抑制演算子

これらの演算子を使うと，シンボル名や個々の文字を修飾して，文字から特別の意味を取り除いたり，引数が評価される時点を制御したりすることができます。

&

機能　代入演算子

モード　MASM, Ideal

構文　&*name*

説明　*name*は，マクロの呼び出しまたは繰り返しブロックの中の実パラメータの値です。多くの場合，パラメータの代入は自動的に行なわれますので，この演算子を使う必要はありません。引用文字列の中で代入したいとき，および1つまたは複数のパラメータを通して渡されるシンボルと何かある文字列を連結するときは，パラメータの前にこの演算子を入れなければなりません。&は，文字列が単一の名前として解釈されるのを防ぐ働きをします。

例

```
MAKEMSG   MACRO   NAME,NUM
MSG&NUM:  DB      '&NAME'
ENDM
```

このマクロを

```
MAKEMSG 9,<Enter a value: >
```

で呼び出すと，次のように展開されます。

```
MSG9:  DB  'Enter a vlue:'
```

<>

機能　文字列演算子

モード　MASM, Ideal

構文　<*text*>

説明　*text*は，1つのマクロパラメータまたは繰り返しパラメータとして扱われます。パラメータを分離するために通常使用されるカンマ，空白，タブを含んでいてもかまいません。これらの分離文字を含む引数を渡したいときは，この演算子を使用しなければなりません。

またこの演算子を使うと，文字の意味を無視して，強制的にリテラルとして扱かわせることができます。たとえば，呼び出したマクロにパラメータとしてセミコロン（;）を渡したいときは，そのセミコロンがコメントの開始として扱われないようにするために不等号カッコ <> で囲まなければなりません。

Turbo Assemblerは，マクロが呼び出されてパラメータが渡されるたびに，外側の1組の不等号カッコを取り除きます。複数レベルのマクロ展開を通してパラメータを渡したいときは，各レベルにつき1組の不等号カッコをつけなければなりません。

例

```
MANYDB MACRO VALS
IRP    X,<VALS>
    ENDM
    ENDM
```

このマクロを呼び出すときは，複数の値は次のように1組の不等号カッコで囲んで，単一のパラメータVALSとして扱われるようにしなければなりません。

```
MANYDB <4,6,0,8>
```

IRP繰り返し疑似命令の中のパラメータの名前も，不等号カッコで囲んであります。これは，マクロを呼び出したときに，マクロに渡すパラメータの両側の不等号カッコが取り除かれるからです。

!

機能　引用文字演算子

モード　MASM, Ideal

構文　!*character*

説明　!演算子を使うと，呼び出したマクロに，マクロ演算子文字を含む引数を渡すことができます。この演算子は，文字を1組の不等号カッコで囲むのと同じような効果があります。たとえば **!&**は＜**&**＞と同じです。

例

```
MAKEMSG    MACRO NAME,NUM
   MSG&NUM: DB '&NAME'
   ENDM
```

上のマクロを次のような引数を指定して

```
KEMSG  <Can't enter !> 99>
```

呼び出すとき，!演算子がなければ引数は途中で切れてしまいます。

%

機能　式評価演算子

モード　MASM, Ideal

構文　%*expression*

説明　*expression*は，この章で説明されている演算子とオペランドからなる数値式または文字列です。マクロを呼び出すときにパラメータとして渡される文字列は，式を評価した結果です。

式そのものではなく，評価された結果を表わす文字列をマクロに渡したいときは，この演算子を使用してください。結果は，現在の基数の数字の文字列で表わされます。また，%の後にテキストマクロの名前を指定することもできます。このようにすると，呼び出したマクロに，パラメータとしてテキストマクロを渡すことができます。

例

```
DEFSYM  MACRO    NUM
    ???&NUM:
    ENDM
```

このマクロを

```
DEFSYM  %5+4
```

で呼び出すと，次のようなコードラベルになります。

```
???9:
```

;;

機能　コメントの抑制

モード　MASM, Ideal

構文　;;*text*

説明　Turbo Assemblerは，2重セミコロン（;;）の後のすべての*text*を無視します。通常のコメントは，マクロ定義の一部として格納され，マクロが展開されるときはいつでもリストの中に現われます。2重セミコロンで始まるコメントは，マクロの定義の一部として格納されません。そのため2重セミコロンを使うとメモリを節約することができます。特にたくさんのコメントを含む多数のマクロがあるときは，効果があります。

例

```
SETBYTES MACRO NAME,VAL
  NAME DB 10 DUP [VAL]    ;;このコメントはマクロの一部として保存されない
  ENDM
```

第3章

疑似命令

アセンブラソースプログラムの命令は，機械命令と疑似命令のどちらかです。機械命令のソース行からは，その命令のニーモニックとオペランドで指定されたプロセッサ用のオブジェクトコードが生成されます。疑似命令のソース行は，データやデータ構造の定義と確保，マクロ定義，リストファイル形式指定，条件アセンブル制御，プロセッサ型と命令セット選択などの命令コードの生成とは無関係な動作を指示します。

疑似命令によってはシンボルを定義するものもあります。そのような疑似命令を使う場合は，ソース行の中にそのシンボル名を指定します。このような疑似命令には，**SEGMENT, LABEL, GROUP**などがあります。その他の疑似命令は，シンボル名を指定しないでアセンブラの動作を指定します。このような疑似命令には，**ORG, IF, %LIST**などがあります。

この章で説明する疑似命令は，アルファベット順になっています（ただし記号は無視します。たとえば**.CODE**は**CODESEG**の前にあります）。

予約キーワードの**%TOC**と**%NOTOC**は，Turbo Assemblerの現在のバージョンでは何も行ないません。将来のバージョンではこれらの予約キーワードを使いますので，プログラム内でこれらのキーワードをシンボルとして使わないでください。

Turbo Assemblerの疑似命令は，次のように3種類に分類できます。

1. ***MASM*方式の疑似命令**

   Turbo Assemblerはすべての***MASM*方式**の疑似命令をサポートします。Turbo AssemblerをIdealモードで使う場合は，これらの疑似命令のうちのいくつかの構文が異なります。このような疑似命令については，両モード（MASMモードとIdealモード）の構文を示します。

2. ***Turbo Assembler*の新しい疑似命令**

   これらの疑似命令は，MASMの疑似命令より機能的に向上しています。

3. **既存の*MASM*疑似命令の同義語となる*Turbo Assembler*疑似命令**

   これらの同義語は，いくつかの既存のMASM疑似命令をより使いやすく再構成したものです。たとえば**.LIST**や**.XLIST**より**%LIST**や**%NOLIST**を使うべきです。一般的

に，ある動作の有効/無効を指定するような一対の疑似命令は，*xxxx*と*NOxxxx*という形式になっています。また同義語では，疑似命令の名前の先頭文字にピリオド(.)を使っていません。ピリオドで始まるMASM疑似命令は，Turbo AssemblerのIdealモードでは使えないので，そのかわりに新しい同義語を使ってください。

リストファイルを制御するすべてのTurbo Assembler疑似命令は，パーセント記号(%)で始まります。

それぞれの疑似命令の構文説明では，次のような記号が使われています。

- ■大カッコ([])は，オプション(任意指定つまり省略可能)の引数を示します。(はタイプする必要はありません)。
- ■省略記号(...)は，直前の項目を必要なだけ何回でも繰り返して指定できることを示します。
- ■**イタリック体**の項目は，プログラムの実際のシンボルや式に置き換えることができます。

サンプル疑似命令

機能　疑似命令の機能の概要です。

モード　疑似命令を使えるモードです。

構文　疑似命令の使い方です。イタリック体の項目は未定義です。

説明　疑似命令の説明です。

参照　関連疑似命令です。

例　疑似命令を使ったサンプルプログラムです。

.186

機能　80186命令のアセンブルを有効にします。

モード　MASM

構文　**.186**

説明　**.186**は，80186プロセッサがサポートする命令のアセンブルを有効にします(**P186**と同様です)。

参照　**.8086, .286, .286C, .286P, .386, .386C, .386P, P8086, P286, P286P, P386, P386P**

例

```
.186
PUSH  1          ; 186では有効
```

.286

機能　80286の非特権命令のアセンブルを有効にします。

モード　MASM

構文　**.286**

説明　**.286**は，80286プロセッサが非特権モードでサポートする命令のアセンブルを有効にします。**.287**または**P287**の疑似命令を指定したときと同様，80287数値演算コプロセッサの命令も有効にします（**P286N**, **.286C**と同様です）。

参照　**.8086, .186, .286C, .286P, .386, .386C, .386P, P8086, P286, P286P, P386, P386P**

例

```
.286
fstsw  ax       ; 80287でのみ有効
```

.286C

機能　80286の非特権命令のアセンブルを有効にします。

参照　.286

.286P

機能　80286のすべての命令のアセンブルを有効にします。

モード　MASM

構文　**.286P**

説明　**.286P**は，80286プロセッサがサポートする（特権モード命令も含む）すべての命令のアセンブルを有効にします。**.287**または**P287**の疑似命令を指定したときと同様，80287数値演算コプロセッサの命令も有効にします（**P286**と同様です）。

参照　**.8086, .186, .286C, .286, .386, .386C, .386P, P8086, P286, P286P, P386, P386P**

.287

機能　80287数値演算コプロセッサの命令のアセンブルを有効にします。

モード　MASM

構文　.287

説明　**.287**は，80287数値演算コプロセッサがサポートするすべての命令のアセンブルを有効にします。この疑似命令は，8087コプロセッサを使ってプログラムを実行することがないとわかっている場合に使います。この疑似命令を使うと浮動小数点命令が8087と互換性のない形式で最適化されるため，8087を使ってプログラムを実行する場合は使わないでください（**.387**と同様です）。

参照　**.8087, .387, P8087, PNO87, P287, P387**

例

```
.287
fsetpm          ; 287でのみ有効
```

.386

機能　80386の非特権命令のアセンブルを有効にします。

モード　MASM

構文　**.386**

説明　**.386**は，80386プロセッサが非特権モードでサポートする命令のアセンブルを有効にします。**.387**または**P387**の疑似命令を指定したときと同様，80387数値演算コプロセッサの命令も有効にします（**P386N**, **.386C**と同様です)。

参照　**.8086, .186, .286C, .286, .286P, .386C, .386P, P8086, P286, P286P, P386, P386P**

例

```
.386
stosd           ; 386でのみ有効
```

.386C

機能　80386の命令のアセンブルを有効にします。

参照　**.8086, .186, .286C, .286, .286P, .386, .386P, P8086, P286, P286P, P386, P386P**

.386P

機能 80386のすべての命令のアセンブルを有効にします。

モード MASM

構文 **.386P**

説明 **.386P**は，80386プロセッサがサポートする（特権モード命令も含む）すべての命令のアセンブルを有効にします。**.387**または**P387**の疑似命令を指定したときと同様，80387数値演算コプロセッサの命令も有効にします（**P386**と同様です）。

参照 **.8086, .186, .286C, .286, .286P, .386C, .386, P8086, P286, P286P, P386, P386P**

.387

機能　80387数値演算コプロセッサの命令のアセンブルを有効にします。

モード　MASM

構文　.387

説明　**.387**は，80387数値演算コプロセッサがサポートするすべての命令のアセンブルを有効にします。この疑似命令は，8087コプロセッサを使ってプログラムを実行することがないとわかっている場合に使います。この疑似命令を使うと浮動小数点命令が8087と互換性のない形式で最適化されるため，8087を使ってプログラムを実行する場合は使わないでください（**.287**と同様です）。

参照　**.8087, .287, P8087, PNO87, P287, P387**

例

```
.387
fsin        ; SIN()は387でのみ有効
```

.8086

機能　8086の命令のみのアセンブルを有効にします。

モード　MASM

構文　**.8086**

説明　**.8086**は，8086の命令のアセンブルを有効にします。80186, 80286, 80386プロセッサに固有の命令を無効にします。8087数値演算コプロセッサの命令は有効にします（**.8087**または**8087**と同様です）。
このモードは，アセンブル開始時のデフォルトの命令セットアセンブルモードです。このモードでアセンブルするプログラムは，80x86プロセッサファミリのすべてのプロセッサで動作します。プログラムが上位プロセッサで動作することがわかっている場合は，そのプロセッサの命令を有効にする疑似命令を指定すれば，その上位命令を利用することができます（**.P8086**と同様です）。

参照　**.186, .286C, .286, .286P, .386C, .386, .386P, P8086, P286, P286P, P386, P386P**

.8087

機能　8087の命令のみのアセンブルを有効にします。

モード　MASM

構文　**.8087**

説明　**.8087**は，8087数値演算コプロセッサの命令のアセンブルを有効にします。80287, 80387プロセッサに固有の命令を無効にします。8087数値演算コプロセッサの命令は有効にします。

このモードは，デフォルトのコプロセッサ命令セットアセンブルモードです。このモードでアセンブルするプログラムは，80x87プロセッサファミリのすべてのコプロセッサで動作します。プログラムが上位コプロセッサで動作することがわかっている場合は，そのプロセッサの命令を有効にする疑似命令を指定すれば，その上位命令を利用することができます（**.P8087**と同様です）。

参照　**.287, .387, P8087, PNO87, P287, P387**

例

```
.8087
fstsw memloc        ; 8087にはFSTSW AX命令はない
```

:

機能 nearコードにラベルを付けます。

モード MASM, Ideal

構文 *name*:

説明 *name*はソースファイル内で初めて定義するシンボルです。nearコードラベルは，1行に単独で記述することも命令の前に記述することもできます。通常nearコードラベルは，**JMP**または**CALL**命令の（同一セグメント内の）ジャンプ先として使うことができます。

PUBLIC疑似命令によって他のソースファイルからアクセス可能になっていない限り，コードラベルはカレントソースファイルの中からのみアクセスすることができます。

この疑似命令は，**LABEL**疑似命令を使って**NEAR**ラベルを定義することと同じです。たとえば，

```
A:
```

は

```
A LABEL NEAR
```

と同じです。

参照 **LABEL**

例

```
      .
      .
      .
      jne A           ; 次の命令はスキップする
      inc si
A:                    ; JNEはここへジャンプする
```

=

機能　数値にシンボルをイコール定義します。

モード　MASM, Ideal

構文　*name*＝*expression*

説明　*name*に*expression*の評価結果を割り当てます(同等にします)。*expression*は，評価されて定数またはセグメント内のアドレスになります。*name*は新しいシンボル名または=疑似命令によって以前定義されたシンボル名です。*expression*には，前方参照のシンボル名が含まれてはなりません。

＝疑似命令を使って定義されたシンボルに別の値をイコール定義することもできます。またシンボルをカウンタとして使うことが可能となります（下の例を参照してください)。

＝疑似命令を使って文字列をイコール定義したり，キーワードや命令ニーモニックを再定義することはできません。このような場合は，**EQU**を使ってください。MASMモードでは，＝疑似命令の方が**EQU**疑似命令より動作が予測しやすいので，できるだけ**EQU**のかわりに＝を使うようにするとよいでしょう。

参照　**EQU**

例

```
MASK = 1                    ; ビットマスクを初期化
BittBl  LABEL  BYTE
    REPT 8
    DB   MASK
MASK = MASK * 2             ; ビットを左へシフト
    ENDM
```

ALIGN

機能　ロケーションカウンタの値を2の累乗に切り上げます。

モード　MASM, Ideal

構文　**ALIGN** *boundary*

説明　*boundary*は，アラインメントの単位を2の累乗（2, 4，8, ...）で指定します。ロケーションカウンタが*boundary*の倍数（のオフセット）にセットされていない場合は, 1バイト長の**NOP**命令をセグメント内に必要なだけ挿入して, ロケーションカウンタを適切なアドレスに切り上げます。ロケーションカウンタが*boundary*の倍数になっている場合は，この疑似命令は何もしません。

ALIGN疑似命令で指定されたセグメントアラインメントよりも単位の大きなアラインメントを行なうことはできません。セグメントアラインメントは, **SEGMENT**疑似命令によってセグメント開始時に指定されます。

次に例を示します。

```
CODE SEGMENT PARA PUBLIC
```

上のようにセグメントを定義した場合は，**ALIGN 16**（**PARA**と同様）は使えますが，**ALIGN 32**は使えません。なぜなら，**ALIGN 32**は**SEGMENT**疑似命令の**PARA**キーワードで指定するアラインメントよりも単位が大きいからです。

セグメントアラインメントの単位が適切（十分）でない場合は，警告メッセージが表示されます。

参照　**EVEN, EVENDATA**

例

```
ALIGN   4               ; 386用のDWORD境界アラインメント
BigNum  DD  12345678
```

.ALPHA

機能　アルファベット順のセグメントオーダ（順序付け）を設定します。

モード　MASM

構文　**.ALPHA**

説明　**.ALPHA**は，通常MASMのかなり前のバージョンとIBMアセンブラとの互換性を保つために使います。これらの旧バージョンアセンブラのデフォルト特性は，セグメントをアルファベット順で生成することです（新しいバージョンでは違います）。旧バージョンのアセンブラ用に書かれたソースファイルのアセンブル時にこの疑似命令を使ってください。

この疑似命令を使わない場合，セグメントの順序はソースファイル内の順序と同じになります。**DOSSEG**疑似命令もセグメントの順序に影響します。

.ALPHAはコマンドラインオプションの/**A**と同じ機能です。/**S**コマンドラインオプションですでにシーケンシャルセグメントオーダを指定してある場合は，**.ALPHA**の指定の方がそのコマンドラインの指定に優先します。

参照　**DOSSEG, SEQ**

例

```
.ALPHA
XYZ   SEGMENT
XYZ   ENDS
ABC   SEGMENT       ; このセグメントが先頭になる
ABC   ENDS
```

ARG

機能　プロシージャに対してスタック上の引数をセットアップします。

モード　MASM, Ideal

構文　**ARG** *argument, argument ... =symbol RETURNS*
argument, argument

説明　**ARG**は，**PROC**と**ENDP**のペアの間に置かれます。この疑似命令によって，プロシージャは呼び出し側プログラムがスタックにプッシュした引数を受け取ることができます。*argument*にはBPレジスタの値に対する正のオフセットが割り当てられます。その際，関数 コールのリターンアドレスと呼び出し側のBPの両方がすでにスタックにプッシュされていることを前提とします。

*argument*は，プロシージャ コールの際に引き渡される引数を示します。**.MODEL**疑似命令で指定された言語によって，引数がスタック上で逆になるかどうかが決ります。プロシージャにおける引数の順番は，プロシージャをコールする高級言語関数における順番と同じでなければなりません。Turbo Assemblerは，必要に応じて（言語に応じて）引数を逆順に読みます。*argument*は次の構文形式を持ちます。

```
argname[:[[distance] PTR] type]
```

*argname*は，プロシージャ内で引数を利用するときに使う引数の名前です。*distance*はオプションで，引数が指定サイズ（**NEAR**または**FAR**）のポインタであることを示します。typeは引数のデータ型で，**WORD, DWORD, FWORD, PWORD, QWORD, TBYTE,** 構造体名，DATAPTRのどれかです。

*type*を省略すると**WORD**が指定されます。

PTRを付けて引数がデータへのポインタであることを示すと，Turbo AssemblerはTurbo Debugger用のデバッグ情報を生成します。つまり，**PTR**を使用してもTurbo Assemblerが生成するコードには影響を与えませんが，追加デバッグ情報の生成に影響を与えます。実際のデータをアクセスするためには，ポインタ引数を使ってコードを書いてください。

PTRの前の**NEAR**または**FAR**を省略すると，カレントメモリモデルに基づいてポ

インタサイズは決まります。ただし386プロセッサについては，カレントセグメントアドレスサイズ（16または32ビット）に基づいてポインタサイズを決めます。
ポインタサイズは，タイニィ，スモール，ミディアムの各メモリモデルでは**WORD**になり，その他の16ビットセグメントを使うメモリモデルでは**DWORD**になります。386を使って32ビットセグメントをアクセスする場合は，ポインタサイズは，タイニィ，スモール，ミディアムの各メモリモデルでは**DWORD**になり，コンパクト，ラージ，ヒュージの各メモリモデルでは**FWORD**になります。
*argument*名変数は，BPに基づくメモリオペランドとしてプロシージャ内で定義されたままとなります。次に例を示します。

```
Func1 PROC NEAR
      arg A:WORD,B:DWORD:4,C:BYTE = D
```

この例では，Aが［*BP+4*］，Bが［*BP+6*］，Cが［*BP+14*］，Dが20としてそれぞれ定義されます。
引数リストの最後に等号（=）とシンボル（*symbol*）を指定すると，*symbol*は引数ブロックの合計サイズ（バイト）と同等の値を持つことになります。この値をプロシージャの最後の**RET**命令に対する引数として使うと，スタックにプッシュされた引数をリターン直前にクリアすることができます（これはPascalの呼び出し規定です）。
8086プロセッサファミリではバイトサイズの引数をプッシュすることはできないので，**BYTE**型で宣言された引数はスタック上で2バイトを占めます。これは，高級言語における文字変数パラメータの扱い方に一致します。引数をスタック上で1バイトに指定したい場合は，次のようにカウントフィールドを指定してください。

```
ARG REALBYTE:BYTE:1
```

この**BYTE**引数のカウント値を省略すると，2が適用されます。
オプションの**RETURNS**キーワードで指定したら，1つまたは複数の引数をさらに追加指定してください。これらの引数は，プロシージャが呼び出し側プログラムへリターンする際にスタックからポップされません。**.MODEL**疑似命令を使って**PASCAL, TPASCAL**などの言語を指定すると，通常はプロシージャからのリターン時にすべての引数がポップされます。**RETURNS**キーワードの後に引数を付けると，呼び出し側プログラムが利用できるように引数はスタック上に残されます。特にPascalの文字列のリターン値を定義するには，**RETURNS**キーワードの後にそれを指定してください。

参照　LOCAL, PROC, USES

例　Pascalのプロシージャ例

```
fp  PROC FAR
    ARG  SRC:WORD,DEST:WORD = ARGLEN
    push bp
    mov  bp,sp
    mov  di,DEST
    mov  si,SRC
; プロシージャの本体
    pop  bp
    ret  ARGLEN
fp  ENDP
```

ASSUME

機能　セグメントレジスタとセグメント名またはグループ名の関係を示します。

モード　MASM, Ideal

構文　**ASSUME** *segmentreg:name, segmentreg:name ...*
ASSUME *segmentreg:NOTHING*
ASSUME *NOTHING*

説明　*segmentreg*は，CS, DS, ES, SSレジスタのどれかです。**P386**または**P386N**疑似命令で80386プロセッサを有効にしてある場合は，FSとGSレジスタも含まれます。*name*は次のどれかです。

■**GROUP**疑似命令を使って定義されたグループの名前
■**SEGMENT**疑似命令または簡略セグメント疑似命令を使って定義されたセグメントの名前
■**SEG**演算子で始まる式
■**NOTHING**キーワード

NOTHINGキーワードは，指定したセグメントレジスタとセグメント名またはグループ名との関係を取り消します。**ASSUME NOTHING**疑似命令は，セグメントレジスタとセグメント名またはグループ名とのすべての関係を解除します。
1つの**ASSUME**疑似命令で複数のレジスタを設定することができます。またソースファイル内には複数のASSUME疑似命令を置くことができます。
ASSUME疑似命令の使い方の詳細については，**ユーザーズガイド**の第10章の「ASSUME疑似命令」の節を参照してください。

参照　**GROUP, SEGMENT**

例

```
DATA    SGMENT
mov   ax,DATA
mov   ds,ax
ASSUME ds:DATA
```

%BIN

機能　リストファイルのオブジェクトコードフィールドの幅を設定します。

モード　MASM, Ideal

構文　**%BIN** *size*

説明　*size*は桁数を指定する定数です。この疑似命令を使わない場合は，リストファイル内の命令のオペコードフィールドは20桁となります。

例

```
%BIN 12      ; リスト幅を12桁に設定
```

CATSTR

機能　2つ以上の文字列を連結して，1つの文字列にします。

モード　MASM51, Ideal

構文　*name* **CATSTR** *string* [*,string*] ...

説明　*name*は結合後の文字列の名前です。
*string*は次のどれかです。

■不等号カッコで囲まれた文字列引数（例　*abc*　）
■定義済みテキストマクロ
■パーセント（%）で始まる式（評価されて数値文字列となる）

参照　**SUBSTR, INSTR, SIZESTR**

例

```
LETTERS  CATSTR  <abc>,<def>   ; LETTERS = "abcdef"
```

.CODE

機能　コードセグメントの開始を定義します。

モード　MASM

構文　**.CODE** *name*

説明　**.CODE**疑似命令は，モジュール内の実行可能コードの開始を指定します。最初に**.MODEL**疑似命令を使ってメモリモデルを指定する必要があります。ミディアムまたはラージメモリモデルを指定した場合は，**.CODE**疑似命令の後にオプションの*name*を指定してセグメント名を示すことができます。1つのファイル内で異なる名前の複数のコードセグメントを指定することができます。

1つのソースファイル内で必要なだけ何回でも**.CODE**疑似命令を指定することができます。ソース内に同一名のセグメントがある場合は，結合して1つのコードセグメントになります。これは，1つの**.CODE**疑似命令の後に一括してコードを入力することと同じです。

.CODE疑似命令を使うと，CSレジスタでカレントコードセグメントをアクセスすることができます。これは，ソースファイル内の**.CODE**疑似命令の後に次の疑似命令を入れて，コードセグメントを指定することと同じです。

```
ASSUME  CS:@code
```

参照　**CODESEG, .DATA, .FARDATA, .FARDATA?, .MODEL, .STACK, .DATA, .FARDTA, .FARDATA?**

例

```
.CODE           ; コードセグメントの開始
mov  al,X
.DATA           ; データセグメントの開始
X DB ?
```

CODESEG

機能　コードセグメントの開始を指定します。

モード　MASM, Ideal

説明　**.CODE**と同様です。

参照　**.CODE, .DATA, .FARDATA, .FARDATA?, .MODEL, .STACK**

COMM

機能　パブリック変数を定義します。

モード　MASM, Ideal

構文　**COMM** *definition* [,*definition*] ...

説明　*definition*はシンボル指定で，次の形式です。

```
[distance] name:type [:count]
```

*distance*はオプションで，**NEAR**または**FAR**のどちらかです。このオプションは，共用変数がnearデータ領域の一部（**DGROUP**）または自身のfarセグメントを占有するかを指定します。*distance*を省略すると，デフォルトのデータメモリモデルのサイズが指定されます。簡略セグメント疑似命令（**.MODEL**など）を使わない場合は，デフォルトサイズは**NEAR**となります。タイニィ，スモール，ミディアムの各モデルでも，デフォルトサイズは**NEAR**となります。他のモデルでは**FAR**となります。

*name*は共用シンボル名であり，その領域はリンク時に確保されます。*type*は，**BYTE, WORD, DWORD, FWORD, PWORD, QWORD, TBYTE,**構造体名のどれかです。

オプションの*count*は，この共用シンボルが定義する項目数を指定します。*count*を省略すると1が指定されます。共用変数用に確保された領域の合計サイズは，*type*フィールドで指定された長さの*count*倍となります。

カンマ（,）で*definition*を区切れば，複数の共用シンボルを指定することができます。

共用変数はリンカによって確保されます。

MASMモードでは，セグメントの外で宣言された共用変数はDSレジスタによってアクセスできることを前提としています。しかし常にアクセスできるとは限りません。

これらの変数を参照する際は，正しいセグメント値がDSにストアされていること，またはセグメントオーバライドが指定されていることを確認してください。Idealモードでは，**ASSUME**疑似命令で指定されたようにカレントセグメントレジスタ

を使って，Turbo Assemblerは共用変数がアドレシング可能であるかどうかをチェックします。

共用変数は初期化することはできません。モジュール間で共用されるデータを初期化する場合は，**GLOBAL**疑似命令を使います。リンカが共用変数を特定の順序で確保する保証はないので，ソース行の**COMM**疑似命令における並びと同じ順番でデータが確保されるとは限りません。

参照　**EXTRN, GLOBAL, PUBLIC**

例

```
COMM buffer:byte:512    ; リンク時に確保される
```

COMMENT

機能　コメントブロックを開始します。

モード　MASM

構文　**COMMENT** *delimiter* [*text*]
[*text*]
delimiter

説明　**COMMENT**は，最初のデリミタ（区切り文字）から次のデリミタを含む行までにあるすべてのテキストを無視します。*delimiter*は**COMMENT**疑似命令の後の最初の空白以外の文字です。

例

```
COMMENT *
    Any old stuff
*
```

%CONDS

機能　条件ブロック内のすべての命令をリスト表示します。

モード　MASM, Ideal

構文　**%CONDS**

説明　リスト制御疑似命令のどれも指定しない場合は，**%CONDS**がデフォルトの条件リストモードになります（**.LFCOND**と同様です)。
%CONDSの指定はリスト制御疑似命令の設定に優先します。リスト制御が有効になっている場合は，条件ブロック内のすべての命令，および偽と評価されたために無視されたブロック内のすべての命令がリスト表示されます。

参照　**%NOCONDS, .LFCOND, .SFCOND, .TFCOND**

例

```
%CONDS
IF 0
  mov ax,1  ; "IF 0"で条件アセンブルが指定されていてもリスト表示される
ENDIF
```

.CONST

機能　定数データセグメントを定義します。

モード　MASM

構文　**.CONST**

説明　.CONST疑似命令は，定数データを入れるプログラム内のセグメントの開始を指定します。この定数データは，プログラムが参照するだけで変更されないデータです。このセグメントには，プロンプトやメッセージ文字列などを入れることができます。
アセンブラだけのプログラムを作る場合は，この疑似命令を使う必要はありません。定数データセグメントを利用すれば，高級言語とリンクするルーチンを書いたり，定数データの初期化を行なうことができます。

参照　**.CODE, .DATA, .DATA?, .FARDATA, .FARDATA?, .MODEL**

例

```
.CONST
msg  DB "Execution terminated"
```

CONST

機能　定数データセグメントを定義します。

モード　MASM, Ideal

参照　**.CODE, .CONST, .DATA, .DATA?, .FARDATA, .FARDATA?**

.CREF

機能　クロスリファレンスリスト（CREF）の表示を有効にします。

モード　MASM

構文　**.CREF**

説明　**.CREF**を指定すると，ソースファイルの以降のすべてのシンボルに対するクロスリファレンス情報を蓄積します。この疑似命令の機能は，クロスリファレンス情報の蓄積を中止する**%XCREF**または**.XCREF**の機能の反対です。
クロスリファレンス情報は，MASMでは別のファイルとして生成されますが，Turbo Assemblerではリストファイルに入れられます。

参照　**%CREF**

%CREF

機能　クロスリファレンスリスト（CREF）の表示を有効にします。

モード　MASM, Ideal

構文　**%CREF**

参照　**%NOCREF, .CREF, .XCREF, %CREFALL, %CREFREF, %CREFUREF**

例

```
%CREF
wval DW 0     ; CREFによってWVALはクロスリファレンスリストに表示される
```

%CREFALL

機能　クロスリファレンスのすべてのシンボルをリスト表示します。

モード　MASM, Ideal

構文　**%CREFALL**

説明　**CREFALL**の機能は，下線記号の付いたシンボルまたは参照シンボルのリスト表示を無効にする**%CREFREF**または**%CREFUREF**の反対です。**%CREFALL**を指定すると，ソースファイル内の後続のすべてのシンボルはクロスリファレンスリストに表示されます。
ソースファイルのアセンブル時はこのモードになっています。

参照　**%CREFREF, %CREFUREF**

例

```
%CREFREF
arg1 equ [bp+4]      ; 参照されていないのでリストされない
%CREFALL
arg2 equ [bp+6]      ; 参照されていないのがリストされる
arg3 equ [bp+8]      ; 参照されているでリストされる
mov  ax,arg3
END
```

%CREFREF

機能　クロスリファレンスの非参照シンボルのリストを無効にします。

モード　MASM, Ideal

構文　**%CREFREF**

説明　**%CREFREF**は，定義されただけで参照されていないシンボル(非参照シンボル)をクロスリファレンスから取り除きます。クロスリファレンスを指定した場合は，これらのシンボルは通常のシンボルテーブルにリスト表示されます。

参照　**%CREF, %CREFUREF, %CREFALL**

例

```
%CREF
abc EQU 4       ; クロスリファレンスにリストされない
xyz EQU 1       ; クロスリファレンスにリストされる
mov ax,xyz      ; XYZはリストされる
END
```

%CREFUREF

機能　非参照シンボルのみをクロスリファレンスにリスト表示します。

モード　MASM, Ideal

構文　**%CREFUREF**

説明　**%CREFUREF**は，シンボルテーブルのクロスリファレンス内に非参照シンボルのリスト表示を有効にします。この疑似命令を使うと，シンボルテーブルは非参照シンボルのみリスト表示されます。参照シンボルと非参照シンボルの両方をリストしたい場合は，**%CREFALL**疑似命令を使ってください。

参照　**%CREFALL, %CREFREF**

例

```
%CREF
abc   EQU 2        ; リスト表示されない
%CREFUREF
def   EQU 1        ; リスト表示される
END
```

%CTLS

機能　リスト制御疑似命令のリスト表示を有効にします。

モード　MASM, Ideal

構文　%CTLS

説明　%CTLSは，リスト制御疑似命令（%LIST, %INCLなど）をリストファイル内に入れることを指定します。これらの疑似命令は，通常はリストされません。%CTLSを指定すると，次の行からリスト制御疑似命令のリスト表示が有効となります。したがって%CTLS自身はリストファイル内に入りません。

参照　%NOCTLS

例

```
%CTLS
%NOLIST     ; この疑似命令はリストファイルに入る
```

.DATA

機能　データセグメントの開始を指定します。

モード　MASM

構文　**.DATA**

説明　**.DATA**疑似命令は，モジュール内に初期化データの開始を定義します。この疑似命令を使うためには，あらかじめ**.MODEL**疑似命令でメモリモデルを指定しておいてください。

.DATA疑似命令は，ソースファイル内で必要なだけ何回でも使うことができます。複数のデータセグメントは1つのデータセグメントに結合されます。これは，1つの**.DATA**疑似命令で一括してデータセグメントを定義したことと同じです。

データセグメントは**DGROUP**と呼ばれるグループに入ります。このグループ内には，**.STACK, .CONST, .DATA?**疑似命令で定義されたセグメントが含まれます。これらのセグメントは，任意のセグメントレジスタで**DGROUP**をポイントすることによってアクセスすることができます。

データセグメントのセグメント属性の詳細については，**.MODEL**疑似命令を参照してください。

参照　**.CODE, .CONST, .DATA , DATASEG, .FARDATA, .FARDATA , .MODEL, .STACK**

例

```
.DATA
array1 DB 100 DUP (0)      ; 初期化されたNEARデータ
```

.DATA?

機能　初期値を設定しないデータセグメントの開始を定義します。

モード　MASM

構文　**.DATA?**

説明　**.DATA?**疑似命令は，モジュール内に初期値を設定しないでデータを定義します。この疑似命令を使うためには，あらかじめ**.MODEL**疑似命令でメモリモデルを指定してください。

次のように，**DUP**演算子を使って初期値を設定しないでデータを定義することができます。

```
DB 6 DUP (?)
```

アセンブラだけのプログラムを作る場合は，この疑似命令を使う必要はありません。この疑似命令を利用すれば，高級言語とリンクするルーチンを書いたり，初期値を設定しないで定数データを定義することができます。

.DATA?疑似命令は，ソースファイル内で必要なだけ何回でも使うことができます。複数のデータセグメントは1つのデータセグメントに結合されます。これは，1つの**.DATA?**疑似命令で一括してデータセグメントを定義したことと同じです。

データセグメントは**DGROUP**と呼ばれるグループに入ります。このグループ内には，**.STACK, .CONST, .DATA**疑似命令で定義されたセグメントが含まれます。

初期値を設定しないデータセグメントのセグメント属性の詳細については，**.MODEL**疑似命令を参照してください。

参照　**.CODE, .CONST, .DATA, .FARDATA, .FARDATA?, .MODEL, .STACK**

例

```
.DATA?
temp DD 4 DUP (?)      ; 初期値を設定しないでデータを定義
```

DATASEG

機能　データセグメントの開始を定義します。

モード　MMASM, Ideal

構文　**DATASEG**

説明　**DATASEG**は，**.DATA**と同様です。

参照　**.CODE, .CONST, .DATA, .DATA?, .FARDATA, .FARDATA?, .MODEL, .STACK**

DB

機能　バイトサイズのメモリ領域を確保します。

モード　MASM, Ideal

構文　[*name*]　**DB** *expression* [*,expression*] ...

説明　*name*は，以降のデータ参照のために使うシンボルです。これを省略してもメモリ領域は確保されますが，シンボル名による参照はできません。

*expression*は，単一または複数のバイトを確保します。次のどれかです。

■-128～＋127, あるいは0～＋255の範囲の定数を表わす(8ビットで表わせる範囲)
■特定の値を与えないで領域を確保する初期値不設定記号の疑問符（？）
■1文字または複数文字から構成される文字列
■**DUP**演算子を使った繰り返し式

参照　**DD, DF, DP, DQ, DT, DW**

例

```
fibs DB 1,1,2,3,5,8,13
buf  DB 80 DUP (?)
msg  DB "Enter value: "
```

DD

機能 ダブルワードサイズのメモリ領域を確保します。

モード MASM, Ideal

構文 [*name*] **DD** [*type PTR*] *expression* [,*expression*] ...

説明 *name*は，以降のデータ参照のために使うシンボルです。これを省略してもメモリ領域は確保されますが，シンボル名による参照はできません。

type **PTR**はオプションです。このオプションは，定義シンボルに対するデバッグ情報を生成し，Turbo Debuggerがその内容を表示します。Turbo Assemblerが生成するデータには，何も影響しません。*type*は，**BYTE, WORD, DWORD, FWORD, PWORD, QWORD, TBYTE,**構造体名，DATAPTRのどれかです。次に例を示します。

```
person  STRUC
name  db  32 dup(?)
age   dw  ?
person  ENDS
pptr  dd person ptr 0     ; PPTRは構造体へのFARポインタ
```

*expression*は，単一または複数のダブルワード(4バイト)を確保します。次のどれかです。

■-2,147,483,648～+2,147,483,647，あるいは0～+4,294,967,295の範囲の定数を表わす（32ビットで表わせる範囲）

■short（32ビット）の浮動小数点数

■特定の値を与えないで領域を確保する初期値不設定記号の疑問符（?）

■16ビットセグメント内のfarアドレス(セグメント オフセット)または32ビットセグメント内のnearアドレス（32ビットのオフセットのみ）を指定するアドレス式

■**DUP**演算子を使った繰り返し式

参照 **DB, DF, DP, DQ, DT, DW**

例

```
Data32 SEGMENT USE32
Xarray DB  0,1,2,3
Data32 ENDS
Data   SEGMENT
Consts DD 3.141, 2.718        ; 浮動小数点定数
DblPtr DD consts              ; 16ビットfarポインタ
NrPtr  DD xarray              ; 32ビットnearポインタ
BigInt DD 12345678            ; large型の整数
Darray DD 4 DUP (1)           ; 4つの整数
```

%DEPTH

機能　リストファイルのdepthフィールドの幅を設定します。

モード　MASM, Ideal

構文　**%DEPTH** *width*

説明　*width*は，リストファイル内の*depth*フィールドの幅を桁数で指定します。*depth*フィールドは，INCLUDEファイルとマクロ命令の展開におけるネスティングレベルを示します。*width*に0を指定すると，このフィールドはリストファイルから除かれます。通常は，2桁あれば最大99レベルを表示できるため，2を超えるフィールド幅を指定する必要はありません。

このフィールド幅のデフォルト値は1です。

例

```
%DEPTH 2        ; 2桁のネストレベルを表示
```

DF

機能　48ビットポインタ（6バイト）のfarデータ領域を定義します。

モード　MASM, Ideal

構文　[*name*]　**DF** [*type PTR*] *expression* [*,expression*] ...

説明　*name*は，以降のデータ参照のために使うシンボルです。これを省略してもメモリ領域は確保されますが，シンボル名による参照はできません。

type **PTR**はオプションです。このオプションは，定義シンボルに対するデバッグ情報を生成し，Turbo Debuggerがその内容を表示します。Turbo Assemblerが生成するデータには，何も影響しません。*type*は，**BYTE, WORD, DWORD, FWORD, PWORD, QWORD, TBYTE,**構造体名，DATAPTRのどれかです。次に例を示します。

```
person  STRUC
name   db  32 dup(?)
age    dw  ?
person  ENDS
DATA    SEGMENT USE32
pptr   df person ptr 0     ; PPTRは構造体への32ビットFARポインタ
```

*expression*は，単一または複数の48ビットfarポインタ（6バイト）を確保します。次のどれかです。

- -140,737,488,355,328～+140,737,488,355,327，あるいは0 281,474,976,710,655の範囲の定数を表わす
- 特定の値を与えないで領域を確保する初期値不設定記号の疑問符（？）
- 48ビットセグメント内のfarアドレス（セグメント：48ビットオフセット）を指定するアドレス式
- **DUP**演算子を使った繰り返し式

この疑似命令は，通常80386プロセッサに対してのみ使うことができます。

参照　**DB, DD, DP, DQ, DT, DW**

例

```
.386
Data  SEGMENT USE32
Msg   DB      "All done"
FmPtr DF      MSG             ; MSGへのFARポインタ
Data  ENDS
```

DISPLAY

機能　引用符で囲まれた文字列を画面に表示します。

モード　MASM, Ideal

構文　**DISPLAY** *"text"*

説明　*text*は，表示したいメッセージです。引用符（"　"）で囲む必要があります。メッセージは，標準出力装置（通常は画面）に出力されます。DOSのリダイレクト機能を使えば，必要に応じて出力先を画面からファイルに切り替えることもできます。

次のように，メッセージ内に代用文字列（&）を入れることもできます。

```
MAKE_DATA MACRO VALUE
          DISPLAY "Initializing a byte to: &VALUE&"
          DB VALUE
          ENDM
```

この疑似命令を使って，条件アセンブルブロックにコメントを付けることができます。

参照　**%OUT**

例　（リスト1行 R3_076.lst）

DOSSEG

機能　リンク時のDOSセグメントオーダ（順序づけ）を有効にします。

モード　MASM, Ideal

構文　**DOSSEG**

説明　**DOSSEG**は，簡略セグメント疑似命令をセットアップする**.MODEL**疑似命令と関連して使います。**DOSSEG**は，プログラム内のセグメントの順序を高級言語におけるセグメントの順序と同じにすることをリンカに指示します。この疑似命令は，スタンドアロンのアセンブラプログラムを作る場合にのみ使い，またプログラムの開始アドレスを指定するメインモジュール内で1回だけ使います。

実行可能プログラムにおけるセグメントの順序は，次のようになります。

1. クラス名が'CODE'のすべてのセグメント
2. クラス名が'CODE'以外の'DGROUP'というグループに属さないすべてのセグメント
3. **DGROUP**に属するすべてのセグメント（以下の順序となる）
   a. クラス名が'BEGDATA'のすべてのセグメント
   b. クラス名が'BEGDATA', 'BSS', 'STACK'のすべてのセグメント
   c. クラス名が'BSS'のすべてのセグメント
   d. クラス名が'STACK'のすべてのセグメント

参照　**.MODEL**

例

```
DOSSEG
.MODEL medium
```

DP

機能　48ビットポインタ（6バイト）のfarデータ領域を定義します。

モード　MASM, Ideal

参照　**DB, DD, DF, DQ, DT, DW**

DQ

機能　クオッドワード（8バイト）のデータ領域を定義します。

モード　MASM, Ideal

構文　[*name*]　**DQ** *expression* [,*expression*] ...

説明　*name*は，以降のデータ参照のために使うシンボルです。これを省略してもメモリ領域は確保されますが，シンボル名による参照はできません。
*expression*は，単一または複数のクオッドワード(8バイト)を確保します。次のどれかです。

■-2^{63}～$2^{63}-1$の範囲の定数を表わす
■long（64ビット）の浮動小数点数
■特定の値を与えないで領域を確保する初期値不設定記号の疑問符（?）
■**DUP**演算子を使った繰り返し式

参照　**DB, DD, DF, DP, DT, DW**

例

```
HugInt DQ 314159265358979323
BigFit DQ 1.2345678987654321
Qarray DQ 10 DUP (?)
```

DT

機能　10バイトのデータ領域を定義します。

モード　MASM, Ideal

構文　[*name*] **DT** *expression* [,*expression*] ...

説明　*name*は，以降のデータ参照のために使うシンボルです。これを省略してもメモリ領域は確保されますが，シンボル名による参照はできません。
*expression*は，単一または複数の10バイト値を確保します。次のどれかです。

■0～99,999,999,999,999,999,999の範囲のパック10進定数を表わす
■特定の値を与えないで領域を確保する初期値不設定記号の疑問符（?）
■一時的に形式化された10バイトの実浮動小数点数
■DUP演算子を使った繰り返し式

参照　**DB, DD, DF, DP, DQ, DW**

例

```
PakNum DD 123456           ; パック10進数
TempVal DD 0.0000000001    ; 高精度浮動小数点数
```

DW

機能　ワードサイズ（2バイト）のデータ領域を定義します。

モード　MASM, Ideal

構文　[*name*]　**DW**　[*type PTR*]　*expression*　[*,expression*] ...

説明　*name*は，以降のデータ参照のために使うシンボルです。これを省略してもメモリ領域は確保されますが，シンボル名による参照はできません。

type **PTR**はオプションです。このオプションは，定義シンボルに対するデバッグ情報を生成し，Turbo Debuggerがその内容を表示します。Turbo Assemblerが生成するデータには，何も影響しません。*type*は，**BYTE, WORD, DWORD, FWORD, PWORD, QWORD, TBYTE,**構造体名，DATAPTRのどれかです。次に例を示します。

```
narray  dw  100  dup (?)
nptr    dw  word ptr narray    ; nptrはワードへのnearポインタ
```

*expression*は，単一または複数のワード（2バイト）を確保します。次のどれかです。

■-32,768～+32,767～+65,535の範囲の定数を表わす（16ビットで表わせる範囲）
■特定の値を与えないで領域を確保する初期値不設定記号の疑問符（?）
■16ビットセグメント内のnearアドレス（オフセットのみ）を指定するアドレス式
■DUP演算子を使った繰り返し式

参照　**DB, DD, DF, DP, DQ, DT**

例

```
int   DW 12345        ; 16ビット整数
Wbuf  DW 6 DUP (?)    ; 6ワードバッファ
Wptr  DW Wbuf         ; オフセット(WBUFへのポインタ)
```

ELSE

機能　他方の条件アセンブルブロックを開始します。

モード　MASM, Ideal

構文　**IF** *condition*

statements1

[**ELSE**

statements2]

ENDIF

説明　**ELSE**に続く命令は，**IF**文に関連する条件が偽と評価されるとアセンブルされます。つまり，*statement1*または*statement2*のどちらか一方だけがアセンブルされます。

ELSE疑似命令は，（他の**ELSE**疑似命令とペアになっていない）もっとも近くの先行する**IF**文とペアになります。**IF**ブロック内に複数の**ELSE**ブロックを入れることができます。

その場合は，**ELSE**ごとに条件の意味が切り換わります。次に例を示します。

```
IF1
     ;アセンブルされる
ELSE
     ;アセンブルされない
ELSE
     ;アセンブルされる
ENDIF
```

参照　**ENDIF, IF, IF1, IF2, IFB, IFDEF, IFDIF, IFDIFI, IFE, IFIDN, IFIDNI, IFNB, IFNDEF**

例

```
IF LargeModel EQ 1
  les di,addr
ELSE
  lea di,addr
ENDIF
```

ELSEIF

機能　条件式が真のときネストした条件アセンブルブロックを開始します。

モード　MASM, Ideal

構文　**ELSEIF** *expression*

説明　*expression*は，評価されて定数になる式です。前方参照のシンボル名を含みません。*expression*が0以外の値に評価された場合は，**ELSEIF**に先行する条件疑似命令（**IF**など）が偽ならば，条件ブロック内の命令はアセンブルされます。

条件ブロック内には，いくつでも**ELSEIF**疑似命令を入れることができます。**ELSEIF**が真の値を持つ場合は，そのコードブロックがアセンブルされ，**ELSEIF**または**ELSE**で定義された条件ブロックの他の部分はスキップされます。1つの条件ブロック内に**ELSE***xx*の形式の複数の疑似命令を混在させることができます。

参照　**ELSEIF1, ELSEIF2, ELSEIFB, ELSEIFDEF, ELSEIFDIF, ELSEIFDIFI, ELSEIFE, ELSEIFIDN, ELSEIFIDNI, ELSEIFNB, ELSEIFNDEF**

例

```
IF ARGSIZE EQ 1
  mov  al,arg
ELSEIF ARGSIZE EQ 2
  mov  ax,arg
ELSE
  %OUT BAD ARGSIZE
ENDIF
```

EMUL

機能　エミュレーションコプロセッサ命令を生成します。

モード　MASM, Ideal

構文　**EMUL**

説明　Turbo Assemblerは，80X87コプロセッサが実行する実浮動小数点命令を生成します。この疑似命令は，プログラムに浮動小数点のエミュレーションパッケージがインストールされていて，それを使う命令を生成したい場合に使ってください。**EMUL**は, **/e**コマンドラインオプションを指定することと同じ効果を持ちます。ファイルのある部分で実浮動小数点命令を生成し，他の部分でエミュレーション命令を生成したい場合は，**EMUL**と**NOEMUL**疑似命令を組み合わせて使います。

参照　**NOEMUL**

例

```
Finit        ; 実8087コプロセッサ命令
EMUL
Fsave buf    ; エミュレーション命令
```

END

機能　ソースファイルの終了を示します。

モード　MASM, Ideal

構文　**END** [*startaddress*]

説明　*startaddress*はオプションのシンボルまたは式で，プログラムの実行開始アドレスを指定します。複数のソースファイルをリンクしてプログラムを作る場合は，どれか1つのファイル内に*startaddress*を指定します。*startaddress*は，モジュール内のアドレスまたは他のモジュール内で定義され**EXTRN**疑似命令で宣言された外部シンボルです。

ソースファイル内の**END**疑似命令以降のテキストは無視されます。

例

```
.MODEL SMALL
.CODE
start:
;プログラムの本体
END start                ; プログラムのエントリポイントは"START"
THIS LINE IS IGNORED (この行は無視される)
SO IS THIS ONE       (この行も同様)
```

ENDIF

機能　条件アセンブルブロックの終了を示します。

モード　MASM, Ideal

構文　**IF** *condition*
statements
ENDIF

説明　**IF***xxxx*という形式の疑似命令で始まる条件アセンブルブロックは，**ENDIF**疑似命令で終了してください。**IF**ブロックは，255レベルまでネストすることができます。

参照　**ELSE, IF, IF1, IF2, IFB, IFDEF, IFDIF, IFDIFI, IFE, IFIDN, IFIDNI, IFNB, IFNDEF**

例

```
IF DebugMode        ; デバッグモードが0でないとき次のブロックをアセンブルする
  mov ax,0
  call DebugDump
ENDIF
```

ENDM

機能 繰り返しブロックまたはマクロの終了を示します。

モード MASM, Ideal

構文 **ENDM**

説明 **ENDM**疑似命令は，マクロ定義または繰り返しブロックの終了を示します。

参照 **IRP, IRPC, MACRO, REPT**

例

```
IRP  reg,<ax,bx,cx,dx>
push reg
ENDM
```

ENDP

機能　プロシージャの終了を示します。

モード　MASM, Ideal

構文　MASMモード：

[*procname*]　**ENDP**

Idealモード：

ENDP [*procname*]

説明　オプションの*procname*を指定する場合は，プロシージャ定義の先頭の**PROC**疑似命令で指定したプロシージャ名と一致していなければなりません。

Idealモードでは，オプションの*procname*は**ENDP**の後に指定することに注意してください。

ENDPは，プロシージャの呼び出し側プログラムへ戻るための**RET**命令を生成するわけではありません。**RET**命令はプログラム内にコーディングしてください。

参照　**ARG, LOCAL, PROC**

例

```
LoadIt PROC
       ; プロシージャの本体
  ret
LoadIt ENDP
```

ENDS

機能　カレントセグメントの構造体または共用体の終了を示します。

モード　MASM, Ideal

構文　MASMモード：

[*segmentname*] **ENDS**

[*strucname*] **ENDS**

Idealモード：

ENDS [*segmentname*]

ENDS [*strucname*]

説明　**ENDS**は，セグメント，構造体，共用体の終了を示します。オプションの*segmentname*を指定する場合は，**SEGMENT**疑似命令で指定したセグメント名と一致させてください。同様にオプションの*strucname*は，**STRUC**疑似命令で指定した構造体名または**UNION**疑似命令で指定した共用体名と一致していなければなりません。Idealモードでは，オプションの名前は**ENDS**の後に指定することに注意してください。

参照　**SEGMENT, STRUC, UNION**

例

```
DATA SEGMENT            ; データセグメントの開始
Barray DB 10 DUP (0)
DATA ENDS               ; データセグメントの終了，任意のデータを含む
Stat STRUC
Mode DW ?
PTR  DD ?
ENDS                    ; 構造体定義の終了
```

EQU

機能 文字列，エリアス，数値に同等なシンボルを定義します。

モード MASM, Ideal

構文 *name* **EQU** *expression*

説明 *name*に*expression*の評価された結果を割り当てます(同等にします)。*name*は，以前定義されていない新規シンボルでなければなりません。MASMモードでは，**EQU**疑似命令によって文字列と同等なシンボルとして定義されたシンボルだけを再定義することができます。

MASMモードでは，**EQU**疑似命令によって以下の3種類の同等シンボルのどれかを生成することが可能です。

- ■エリアス：キーワードまたは命令ニーモニックを再定義します。エリアスを利用すれば，定義済みのシンボルに対して別名を与えることができます。
- ■式：評価された結果，定数またはアドレスになります。=疑似命令を使った場合も同様です。
- ■文字列：*name*が式中にある場合は，式は後に置き換えられるテキスト文字列としてストアされます。式が評価された結果，エリアス，定数，アドレスのどれにもならない場合は，文字列式となります。

参照 =

例

```
BlkSize EQU 512
BufBlks EQU 4
BufSize EQU BlkSize*BufBlks
BufLen  EQU BufSize                  ; BUFSIZEのエリアス
DoneMsg EQU <'DOSへ戻る'>
```

.ERR

機能　エラーを発生させます。

モード　MASM

構文　**.ERR**

説明　**.ERR**をソースファイル内に入れておくと，その行でエラーメッセージが発生します。

通常この疑似命令は，特定のアセンブル時条件が満足されたかどうかをチェックする条件アセンブルブロック内で使います。

参照　**.ERR1, .ERR2, .ERRE, .ERRNZ, .ERRNDEF, .ERRDEF, .ERRB, .ERRNB, .ERRIDN, .ERRIDNI, .ERRDIF, .ERRDIFI**

例

```
IF $ GT 400h
  .ERR                  ; セグメントが大きすぎる
  %OUT セグメントが大きすぎます.
ENDIF
```

ERR

機能　エラーを発生させます。

モード　MASM, Ideal

構文　**ERR**

説明　**.ERR**と同様です。

参照　**.ERR1, .ERR2, .ERRE, .ERRNZ, .ERRNDEF, .ERRDEF, .ERRB, .ERRNB, .ERRIDN, .ERRIDNI, .ERRDIF, .ERRDIFI**

.ERR1

機能　1パス目でエラーを発生させます。

モード　MASM

構文　**.ERR1**

説明　Turbo Assemblerは1パスアセンブラなので，**.ERR1**は1パス目つまりアセンブリパスでエラーメッセージを表示します。したがってエラーメッセージは画面に表示されるだけで，2パス目に作られるリストファイルには入りません（2パスアセンブラでは，この疑似命令の意味は異なります）。
Turbo Assemblerは1パスアセンブラなので，警告メッセージも表示します。これによって，この疑似命令はこのアセンブラがパスに依存しており，予想どおりに動かないこともあることを知ることができます。

参照　**.ERR2**

例

```
.ERR1        ; この疑似命令はリスト表示されない
```

.ERR2

機能 2パス目でエラーを発生させます。

モード MASM

構文 **.ERR2**

説明 Turbo Assemblerは1パスアセンブラなので，**.ERR2**は2パス目つまりリスティングパスでエラーメッセージを表示します。したがってエラーメッセージはリスト表示されますが，実際のアセンブルプロセスの間は画面に表示されません(2パスアセンブラでは，この疑似命令の意味は異なります)。
Turbo Assemblerは1パスアセンブラなので，警告メッセージも表示します。これによって，この疑似命令はこのアセンブラがパスに依存しており，予想どおりに動かないこともあることを知ることができます。

参照 **.ERR1**

例
```
.ERR2        ; この疑似命令はリストファイルにのみ現われる
```

.ERRB

機能　引数がブランク（空白）ならば，エラーを発生させます。

モード　MASM

構文　**.ERRB** <*argument*>

説明　**.ERRB**は常にマクロ内で使います。指定されたダミー（仮）引数と置き換わるための実引数を伴ってマクロが呼び出されたかどうかをチェックします。***argument***がブランク（空白）の場合は，ソース行のマクロが呼び出された場所でエラーメッセージが発生します。

チェックする引数は，常に不等号カッコ（<>）で囲んでください。

参照　**.ERRNB**

例

```
double MACRO arg1
  .ERRB <arg1>       ; 引数が必要
  shl arg1,1         ; 引数の値を2倍する
ENDM
```

.ERRDEF

機能　シンボルが定義済みならば，エラーを発生させます。

モード　MASM

構文　**.ERRDEF** *symbol*

説明　**.ERRDEF**は，ソースファイル内で*symbol*がすでに定義されていた場合にカレント行番号でエラーメッセージを表示します。

参照　**.ERRNDEF**

例

```
SetMode MACRO ModeVal
    .ERRDEF  _MODE        ; 定義済みならばエラー
_MODE  EQU ModeVal
    ENDM
```

.ERRDIF

機能　2つの引数が異なれば，エラーを発生させます。

モード　MASM

構文　**.ERRDIF** <*argument1*>, <*argument2*>

説明　**.ERRDIF**は常にマクロ内で使い，2つの引数が同じ文字列かどうかをチェックします。同じ文字列でない場合は，ソース行のマクロが呼び出された場所でエラーメッセージが発生します。2つの文字列は，1文字ずつ大文字小文字の区別をつけて比較されます。大文字小文字の区別を無視したい場合は，**.ERRDIFI**疑似命令を使ってください。

各引数は常に不等号カッコ（<>）で囲み，2つの引数はカンマで区切ってください。

参照　**.ERRIDN, .ERRDIFI, .ERRIDNI**

例

```
SegLoad  MACRO reg,val
    .ERRDIF <reg>,<es>      ; ESレジスタのみ有効
    mov ax,val
    mov reg,ax
    ENDM
```

.ERRDIFI

機能　2つの引数が異なれば，エラーを発生させます。大文字小文字の区別は無視します。

モード　MASM

構文　**.ERRDIFI** <*argument1*>, <*argument2*>

説明　**.ERRDIFI**は常にマクロ内で使い，2つの引数が同じ文字列かどうかをチェックします。同じ文字列でない場合は，ソース行のマクロが呼び出された場所でエラーメッセージが発生します。2つの文字列は，1文字ずつ大文字小文字の区別を無視して比較されます。大文字小文字の区別をつけたい場合は，**.ERRDIF**疑似命令を使ってください。

各引数は常に不等号カッコ（＜＞）で囲み，2つの引数はカンマで区切ってください。

参照　**.ERRIDN, .ERRDIF, .ERRIDNI**

例

```
SegLoad  MACRO reg,val
    .ERRDIFI <reg>,<es>       ; ESレジスタのみ有効とする
    mov ax,val                ; REGの大文字小文字の区別を無視する
    mov reg,ax
    ENDM
```

.ERRE

機能　式が偽（0）ならば，エラーを発生させます。

モード　MASM

構文　.ERRE *expression*

説明　*expression*は，評価されて定数にならなければなりません。前方参照のシンボル名を含んではなりません。式が0と評価された場合は，カレントソース行でエラーメッセージが発生します。

参照　.ERRNZ

例

```
PtrLoad  MACRO ptr,val
    .ERRE val          ; ポインタに0をロードしようとするとエラーとなる
    mov si,val
    ENDM
```

.ERRIDN

機能　2つの引数が同じならば，エラーを発生させます。

モード　MASM

構文　**.ERRIDN** <*argument1*>, <*argument2*>

説明　**.ERRIDN**は常にマクロ内で使い，2つの引数が同じ文字列かどうかをチェックします。2つの文字列が同じ場合は，ソース行のマクロが呼び出された場所でエラーメッセージが発生します。2つの文字列は，1文字ずつ大文字小文字の区別をつけて比較されます。大文字小文字の区別を無視したい場合は，**.ERRIDNI**疑似命令を使ってください。

各引数は常に不等号カッコ（<>）で囲み，2つの引数はカンマで区切ってください。

参照　**.ERRDIF, .ERRIDNI, .ERRDIFI**

例

```
PushSeg  MACRO reg,val
    .ERRIDN <reg>,<cs>        ; CSレジスタへのロードは不正とする
    push reg
    mov reg,val
    ENDM
```

.ERRIDNI

機能 2つの引数が同じならば，エラーを発生させます。大文字小文字の区別は無視します。

モード MASM

構文 **.DRRIDNI** <*argument1*>, <*argument2*>

説明 **.ERRIDNI**は常にマクロ内で使い，2つの引数が同じ文字列かどうかをチェックします。2つの文字列が同じ場合は，ソース行のマクロが呼び出された場所でエラーメッセージが発生します。2つの文字列は，1文字ずつ大文字小文字の区別を無視して比較されます。大文字小文字の区別をつけたい場合は，**.ERRIDN**疑似命令を使ってください。

各引数は常に不等号カッコ（<>）で囲み，2つの引数はカンマで区切ってください。

参照 **.ERRDIF, .ERRIDN, .ERRDIFI**

例

```
PushSeg  MACRO reg,val
    .ERRIDN <reg>,<cs>        ; CSレジスタへのロードは無効
    push reg                  ; CSとcsは区別しない
    mov reg,val
    ENDM
```

ERRIF

機能　式が真（0以外）ならば，エラーを発生させます。

モード　MASM, Ideal

参照　**.ERRE, .ERRNZ**

ERRIF1

機能　1パス目でエラーを発生させます。

モード　MASM, Ideal

参照　**.ERR1**

ERRIF2

機能　2パス目でエラーを発生させます。

モード　MASM, Ideal

参照　**.ERR2**

ERRIFB

機能　引数がブランクならば，エラーを発生させます。

モード　MASM, Ideal

参照　**.ERRB**

ERRIFDEF

機能　シンボルが定義済みならば，エラーを発生させます。

モード　MASM, Ideal

参照　**.ERRDEF**

ERRIFDIF

機能　2つの引数が異なれば，エラーを発生させます。

モード　MASM, Ideal

参照　**.ERRDIF**

ERRIFDIFI

機能 2つの引数が異なれば，エラーを発生させます。大文字小文字の区別を無視します。

モード MASM, Ideal

参照 **.ERRDIFI**

ERRIFE

機能 式が偽（0）ならば，エラーを発生させます。

モード MASM, Ideal

参照 **.ERRE**

ERRIFIDN

機能 2つの引数が同じならば，エラーを発生させます。

モード MASM, Ideal

参照 **.ERRIDN**

ERRIFIDNI

機能　2つの引数が同じならば，エラーを発生させます。大文字小文字の区別は無視します。

モード　MASM, Ideal

参照　**.ERRIDNI**

ERRIFNB

機能　引数がブランクでないならば，エラーを発生させます。

モード　MASM, Ideal

参照　**.ERRNB**

ERRIFNDEF

機能　シンボルが未定義ならば，エラーを発生させます。

モード　MASM, Ideal

参照　**.ERRNDEF**

.ERRNB

機能　引数がブランクでなければ，エラーを発生させます。

モード　MASM

構文　**.ERRNB** <*argument*>

説明　**.ERRNB**は常にマクロ内で使います。指定されたダミー（仮）引数と置き換わるための実引数を伴ってマクロが呼び出されたかどうかをチェックします。*argument*がブランク（空白）でない場合は，ソース行のマクロが呼び出された場所でエラーメッセージが発生します。
チェックする引数は，常に不等号カッコ（<>）で囲んでください。

参照　**.ERRB**

例

```
DoIt MACRO a,b
    .ERRNB <B>          ; 引数が1つだけ必要
    ;
    ENDM
```

.ERRNDEF

機能　シンボルが未定義ならば，エラーを発生させます。

モード　MASM

構文　**.ERRNDEF** *symbol*

説明　**.ERRDEF**は，ソースファイル内で*symbol*がまだ定義されていない場合にカレント行番号でエラーメッセージを表示します。前方参照シンボルは，エラーとなります。何故なら，前方参照シンボルを見つけた時点では，そのシンボルはまだ定義されていないからです。

参照　**.ERRDEF**

例

```
.ERRNDEF BufSize    ; この時点ではBufSizeというシンボルは未定義なのでエラー
buf DB BufSize
```

.ERRNZ

機能 式が真（0以外）ならば，エラーを発生させます。

モード MASM

構文 **.ERRNZ** *expression*

説明 *expression*は，評価されて定数にならなければなりません。前方参照のシンボル名を含んではなりません。式が0以外に評価された場合は，カレントソース行でエラーメッセージが発生します。

参照 **.ERRE**

例

```
.ERRNZ $ GT 1000h      ; セグメントが大きすぎる場合はエラー
```

EVEN

機能　ロケーションカウンタを次の偶数アドレスまで切り上げます。

モード　MASM, Ideal

構文　**EVEN**

説明　**EVEN**は，16ビットデータバスを使うプロセッサ（8086, 80186, 80286）が効率よくアクセスできるようにコードのアラインメント（境界合わせ）を行ないます。8ビットデータバスを使うプロセッサ（8088, 80188）の性能を向上させるものではありません。
この疑似命令は，セグメントを開始する**SEGMENT**疑似命令内で指定することはできますが，**BYTE**アラインメントを使うこのセグメント内で使うことはできません。
EVEN疑似命令を見つけたときロケーションカウンタが奇数の場合は，1バイトの**NOP**命令をセグメント内に挿入してロケーションカウンタを偶数にします。このようにNOP命令を詰め込むことによって，**EVEN**をコードセグメント内で使っても実行時にエラー命令が実行されることはありません。ロケーションカウンタがすでに偶数のときは，この疑似命令は何もしません。
セグメントアラインメントの単位が十分大きくない場合は，**EVEN**疑似命令の警告メッセージが表示されます。

参照　**ALIGN, EVENDATA**

例

```
EVEN
@A:   LODSB
      XOR    bl,al    ; 効率のよいアクセスのためのアラインメント
      LOOP   @A
```

EVENDATA

機能　ロケーションカウンタをデータセグメントの次の偶数アドレスまで切り上げます。

モード　MASM, Ideal

構文　**EVENDATA**

説明　**EVENDATA**は，16ビットデータバスを使うプロセッサ（8086, 80186, 80286）が効率よくアクセスできるようにコードのアラインメントを行ないます。8ビットデータバスを使うプロセッサ（8088, 80188）の性能を向上させるものではありません。

この疑似命令は，セグメントを開始する**SEGMENT**疑似命令内で指定することはできますが，**BYTE**アラインメントを使うこのセグメント内で使うことはできません。

EVENDATA疑似命令を見つけたときロケーションカウンタが奇数の場合は，1バイトの0をセグメント内に挿入してロケーションカウンタを偶数にします。ロケーションカウンタがすでに偶数のときは，この疑似命令は何もしません。

この疑似命令は，データを生成しないでロケーションカウンタを進めるだけで偶数アラインメントを行ないます。この方法は，初期値を設定しないセグメントに対しては便利です。また　セグメントアラインメントの単位が十分大きくない場合は，警告メッセージが表示されます。

参照　**ALIGN, EVEN**

例

```
EVENDATA
var1    DW    0    ; 効率のよい8086のアクセスのためのアラインメント
```

EXITM

機能　マクロ命令またはブロックの繰り返しの展開を中止します。

モード　MASM, Ideal

構文　**EXITM**

説明　**EXITM**は，マクロ命令の展開または繰り返しブロックの展開を途中で中止します。**EXITM**以降にある命令は，すべて無視されます。
この疑似命令は，多重レベルの条件アセンブルブロックから出るときに使うと便利です。

参照　**ENDM, IRP, IRPC, REPT, MACRO**

例

```
Shift MACRO op,n
count = 0
  REPT N
  shl OP,N
count = count + 1
  IF count GE 8         ; COUNT≧8ならば展開を中止する
    EXITM
  ENDIF
  ENDM
```

EXTRN

機能　シンボルが他モジュール内で定義されていることを示します。

モード　MASM, Ideal

構文　**EXTRN** *definition* [*,definition*] ...

説明　*definition*はシンボルの記述で，次の形式を持ちます。

```
name:type [:count]
```

*name*は他モジュール内で定義されたシンボルです。*type*は，他モジュール内で定義されたシンボルの型と一致していなければなりません。次の型のどれかです。

■**NEAR, FAR, PROC**のどれか。**PROC**は，**NEAR**または**FAR**です（**MODEL**疑似命令で設定されたメモリモデルによって異なります）。
■**BYTE, WORD, DWORD, FWORD, PWORD, QWORD, TBYTE,**構造体名，**DATAPTR**のどれか。
■**ABS**

オプションの*count*は，この外部シンボルが定義している項目数を指定します。他のファイル内のシンボル定義において，**DUP**疑似命令を使って複数の項目を確保している場合は，*count*にその項目数を指定してください。**SIZE**と**LENGTH**演算子は，この情報に基づいて外部データの適正サイズを決めることができます。*count*を省略した場合は，1が指定されます。
複数のシンボルを定義する場合は，カンマ（,）で各シンボル定義を区切ってください。
プログラムを正しくリンクするためには，*name*が他モジュール内で**PUBLIC**として宣言されていなければなりません。
EXTRN疑似命令は，**SEGMENT**疑似命令で宣言されたセグメントの内側でも外側でも使うことができます。**EXTRN**疑似命令をセグメントの内側に置いた場合は，外部変数は他モジュール内の同一セグメント内にあることをアセンブラに示します。**EXTRN**疑似命令をセグメントの外側に置いた場合は，どのセグメント内

で変数が宣言されたのかが不明であることをアセンブラに示します。

MASMモードでは，セグメントの外側で宣言された外部シンボルは，DSレジスタを使ってアクセスできることを前提とします（ただし常にアクセス可能とは限りません)。これらの変数を参照する際は，DSレジスタに正しいセグメントの値がセットされていることを確認するか，セグメントオーバライドを指定してください。

Idealモードでは，カレントセグメントレジスタを使って外部変数をアドレシングできるかどうかがチェックされます(**ASSUME**疑似命令の説明を参照してください)。

EXTRN疑似命令の引数の構文は，**ARG**または**LOCAL**疑似命令の引数の構文と同じです。

参照　**COMM, GLOBAL, PUBLIC**

例

```
EXTRN aproc:NEAR
    call aproc      ; 他モジュール内をコールする
```

.FARDATA

機能　farデータセグメントの開始を指定します。

モード　MASM

構文　**.FARDATA** [*name*]

説明　**.FARDATA**は，初期化されたデータセグメントの開始を指定します。複数の分離したfarデータセグメントを作りたい場合は，オプションの*name*でデフォルトセグメント名をオーバライドして，新しいセグメントを作ってください。

.FARDATA疑似命令は，ソースファイル内で必要なだけ何回でも使うことができます。複数のデータセグメントは1つのデータセグメントに結合されます。これは，1つの**.FARDATA**疑似命令で一括してデータセグメントを定義したことと同じです。

farデータセグメントをグループ内に入れることはできません。データをアクセスするには，あらかじめfarセグメントのアドレスをセグメントレジスタにロードすることによって，farセグメントにアクセスできるように指定しておいてください。

farデータセグメントのセグメント属性の詳細については，**.MODEL**疑似命令を参照してください。

参照　**.FARDATA?, .CODE, .DATA, .MODEL, .STACK**

例

```
.FARDATA
FarBuf DB 80 DUP (0)
.CODE
    mov ax,@fardata
    mov ds,ax
    ASSUME ds:@fardata
    mov al,FarBuf[0]        ; バッファの先頭バイトをロード
```

.FARDATA?

機能　初期値を設定しないfarデータセグメントの開始を指定します。

モード　MASM

構文　**.FADATA?** [*name*]

説明　**.FARDATA?**は，初期値を設定しないデータセグメントの開始を指定します。複数の分離したfarデータセグメントを作りたい場合は，オプションの*name*でデフォルトセグメント名をオーバライドして，新しいセグメントを作ってください。

.FARDATA?疑似命令は，ソースファイル内で必要なだけ何回でも使うことができます。複数のデータセグメントは1つのデータセグメントに結合されます。これは，1つの**.FARDATA?**疑似命令で一括してデータセグメントを定義したことと同じです。

farデータセグメントをグループ内に入れることはできません。データをアクセスするには，あらかじめfarセグメントのアドレスをセグメントレジスタにロードすることによって，farセグメントにアクセスできるように指定しておいてください。

初期値を設定しないfarデータセグメントのセグメント属性の詳細については，**.MODEL**疑似命令を参照してください。

参照　**.CODE, .DATA, .FARDATA, .MODEL, .STACK**

例

```
.FARDATA?
FarBuf DB 80 DUP (?)
.CODE
    mov ax,@fardata
    mov ds,ax
    ASSUME ds:@fardata
    mov al,FarBuf(0)        ; バッファの先頭バイトをロード
```

FARDATA

機能　farデータセグメントの開始を指定します。

モード　MASM, Ideal

構文　**FARDATA** [*name*]

説明　**.FARDATA**と同様です。

参照　**.FARDATA**

GLOBAL

機能　グローバルシンボルを定義します。

モード　MASM, Ideal

構文　GLOBAL *definition* [*,definition*] ...

説明　**GLOBAL**疑似命令は，**EXTRN**と**PUBLIC**疑似命令を組み合わせた機能を持ちます。*definition*は，次の形式のシンボル指定です。

```
name:type [:count]
```

*name*がカレントソースファイル内で定義された場合は，*name*は**PUBLIC**疑似命令が指定された場合と同様にパブリックシンボルとなります。*name*がカレントソースファイル内で定義されない場合は，*name*は**EXTRN**疑似命令が指定された場合と同様に（typeで指定された型の）外部シンボルとなります。
*type*は，モジュール内で定義されたシンボルの型と一致しなければなりません。次の型のどれかです。

■**NEAR, FAR, PROC**のどれか。
■**BYTE, WORD, DWORD, FWORD, PWORD, QWORD, TBYTE,**構造体名，DATAPTRのどれか。
■**ABS**

オプションの*count*は、この外部シンボルが定義している項目数を指定します。他のファイル内のシンボル定義において，**DUP**疑似命令を使って複数の項目を確保している場合は，*count*にその項目数を指定してください。**SIZE**と**LENGTH**演算子は，この情報に基づいて外部データの適正サイズを決めることができます。*count*を省略した場合は，1が指定されます。
GLOBAL疑似命令を指定しておけば，すべてのソースファイルがINCLUDEファイルをインクルードすることができます。INCLUDEファイル内には，グローバルシンボルとして定義した共用データのすべてが入っています。これらのデータをモジュールから参照した場合は，**GLOBAL**定義は**EXTRN**疑似命令と同じ機能を

持ち，他モジュール内においてデータがどのように定義されているのかを記述します。データを定義したモジュール内では，**GLOBAL**定義は**PUBLIC**疑似命令と同じ機能を持ち，データは他モジュールから参照可能となります。

複数のパブリックシンボルを定義する場合は，カンマ (,) で各シンボル定義を区切ってください。

ソースファイル内でグローバルシンボルを初めて使う場合は，使う前にシンボルを**GLOBAL**として定義しておいてください。

GLOBAL疑似命令の引数の構文は，**EXTRN**，**ARG**，**LOCAL**疑似命令の引数の構文と同じです。

参照　**COMM, EXTRN, PUBLIC**

例

```
GLOBAL X:WORD,Y:BYTE
X DW 0                  ; 他モジュールに対してパブリックとなる
    mov al,Y            ; Yは外部シンボルとして定義
```

GROUP

機能　複数のセグメントを1つのセグメントレジスタでアクセスできるように定義(グループ化）します。

モード　MASM, Ideal

構文　MASMモード:

name **GROUP** *segmentname* [,*segmentname*] ...

Idealモード:

GROUP *name* *segmentname* [,*segmentname*] ...

説明　*name*はグループの名前を指定します。*segmentname*は, **SEGMENT**疑似命令によってすでに定義したセグメント名または**SEG**で始まる式のどちらかです。*name*は**ASSUME**疑似命令で指定することも，式中の定数としても使うことができます。どちらの場合でも，評価されてグループの先頭パラグラフアドレスになります。

グループ内のすべてのセグメントは，リンク時には連続している必要はありませんが，64Kバイト内に納っている必要があります。

グループの構成セグメント内のシンボルをアクセスする場合は, MASMモードではそのシンボルに対してグループオーバライドを使ってください。Idealモードでは Turbo Assemblerがそのシンボルに自動的にグループオーバライドを付けます。

下に示す例では, *var1*と*var2*は別のセグメントに属していますが, どちらも**DGROUP**というグループに属しています。DSセグメントレジスタに**DGROUP**のベースアドレスを一度設定しておけば, *var1*と*var2*は1つのセグメントに属している変数としてアクセスすることができます。

Idealモードでは, *name*は**GROUP**キーワードの後に指定することに注意してください。

参照　**SEGMENT, ASSUME**

例

```
DGROUP GROUP seg1,seg2
seg1 SEGMENT
     var1 DW 3
seg1 ENDS
seg2 SEGMENT
     var2 DW 5
seg2 ENDS
seg3 SEGMENT
     mov   ax,DGROUP      ; グループのベースアドレスをロード
     mov   ds,ax          ; データをアクセスするためにDSをセット
     ASSUME DS:DGROUP     ; アセンブラにDSの設定を知らせる
     mov   ax,var1
     mul   var2
seg3 ENDS
```

IDEAL

機能　アセンブルモードをIdealモードにします。

モード　MASM, Ideal

構文　**IDEAL**

説明　**IDEAL**疑似命令は, Idealモード用のより厳密かつ型チェックされた構文だけを受け付けることを式パーサに対して指示します。Idealモードの機能と特長の詳細については，**ユーザーズガイド**の第12章を参照してください。

Idealモードは，**MASM**疑似命令または**QUIRKS**疑似命令が指定されるまで有効です。

参照　**QUIRKS, MASM**

例

```
IDEAL
    mov [BYTE PTR DS:SI],1     ; Idealモードのオペランド構文
```

IF

機能　条件アセンブルブロックを開始します。式が真（0以外）の場合にブロックが有効になります。

モード　MASM，Ideal

構文　**IF** *expression*

説明　*expression*は評価されて定数になる式で，前方参照のシンボル名を含みません。*expression*が0以外の値に評価された場合は，条件ブロック内の命令がアセンブルされます。

条件アセンブルブロックを終了するには，**ENDIF**疑似命令を使ってください。

参照　**ENDIF, ELSE, IF1, IF2, IFB, IFDEF, IFDIF, IFDIFI, IFE, IFIDN, IFIDNI, IFNB, IFNDEF**

例

```
IF DoBuffering
    mov ax,BufNum
    call ReadBuf
ENDIF
```

IF1

機能　条件アセンブルブロックを開始します。1パス目にブロックが有効になります。

モード　MASM, Ideal

構文　**IF1**

説明　Turbo Assemblerは1パスアセンブラなので,条件ブロック内の命令はアセンブルパスでのみアセンブルされ,2パス目のリスティングパスではアセンブルされません（2パスアセンブラでは，この疑似命令の意味は異なります)。

前方参照の演算子の再定義を使っている場合は，異常の旨が常にリストファイルに反映されるとは限りません。2パス目でリストファイルが生成された時点では演算子はすでに再定義されているので，リストが一見正しく見えてもオブジェクトファイルのコードが正しく生成されているとは限りません。

条件アセンブルブロックを終了するには，**ENDIF**疑似命令を使ってください。

参照　**ELSE, ENDIF, IF, IF2, IFB, IFDEF, IFDIF, IFDIFI, IFE, IFIDN, IFIDNI, IFNB, IFNDEF**

例

```
IF1
        ; この疑似命令はリストには入らない
ENDIF
```

IF2

機能　条件アセンブルブロックを開始します。2パス目にブロックが有効になります。

モード　MASM, Ideal

構文　**IF2**

説明　Turbo Assemblerは1パスアセンブラなので,条件ブロック内の命令はリスティングパスでのみアセンブルされ,1パス目のアセンブルパスではアセンブルされません（2パスアセンブラでは，この疑似命令の意味は異なります)。

Turbo Assemblerは1パスアセンブラなので，警告メッセージも表示します。これによって，この疑似命令はこのアセンブラがパスに依存しており，予想どおりに動かないこともあることを知ることができます。

前方参照の演算子の再定義を使っている場合は，異常の旨が常にリストファイルに反映されるとは限りません。2パス目でリストファイルが生成された時点では演算子はすでに再定義されているので，リストが一見正しく見えてもオブジェクトファイルのコードが正しく生成されているとは限りません。

条件アセンブルブロックを終了するには，**ENDIF**疑似命令を使ってください。

参照　**ELSE, ENDIF, IF, IF1, IFB, IFDEF, IFDIF, IFDIFI, IFE, IFIDN, IFIDNI, IFNB, IFNDEF**

例

```
IF2
        ; この疑似命令だけがリストに入る
ENDIF
```

IFB

機能　条件アセンブルブロックを開始します。引数がブランクの場合にブロックが有効になります。

モード　MASM, Ideal

構文　**IFB** <*argument*>

説明　*argument*がブランク（空白）の場合は，条件ブロック内の命令がアセンブルされます。**IFB**疑似命令は，指定されたダミー(仮)引数と置き換わるための実引数を伴ってマクロが呼び出されたかどうかのチェックに使ってください。
チェックする引数は，常に不等号カッコ（<>）で囲んでください。
条件アセンブルブロックを終了するには，**ENDIF**疑似命令を使ってください。

参照　**ELSE, ENDIF, IF, IF1, IF2, IFDEF, IFDIF, IFDIFI, IFE, IFIDN, IFIDNI, IFNB, IFNDEF**

例

```
print MACRO msg
  IFB <msg>
    mov si,DefaultMsg
  ELSE
    mov si,msg
  ENDIF
    call ShowIt
  ENDM
```

IFDEF

機能　条件アセンブルブロックを開始します。シンボルが定義済みの場合にブロックが有効になります。

モード　MASM, Ideal

構文　**IFDEF** *symbol*

説明　*symbol*がすでに定義されていた場合は，条件ブロック内の命令がアセンブルされます。

条件アセンブルブロックを終了するには，**ENDIF**疑似命令を使ってください。

参照　**ELSE, ENDIF, IF, IF1, IF2, IFB, IFDIF, IFDIFI, IFE, IFIDN, IFIDNI, IFNB, IFNDEF**

例

```
IFDEF SaveSize
  buf DB SaveSize DUP (?)    ; SAVESIZEが定義されているときのみ
                             ; BUFFERを定義する
  ENDIF
```

IFDIF, IFDIFI

機能　条件アセンブルブロックを開始します。2つの引数が異なる場合にブロックが有効になります。

モード　MASM, Ideal

構文　**IFDIF** <*argument1*>,<*argument2*>

説明　**IFDIF**は常にマクロ内で使い，2つの引数が同じ文字列かどうかをチェックします。どちらかの引数としてマクロ命令のダミー（仮）引数を指定できます。マクロコールへのダミー（仮）引数は，比較実行前にマクロコールへの実引数に置き換えられます。2つの文字列が同じでない場合は，条件ブロック内の命令がアセンブルされます。2つの文字列は，1文字ずつ大文字小文字の区別をつけて比較されます。大文字小文字の区別を無視したい場合は，**IFDIFI**疑似命令を使ってください。

条件アセンブルブロックを終了するには，**ENDIF**疑似命令を使ってください。

参照　**ELSE, ENDIF, IF, IF1, IF2, IFB, IFDEF, IFE, IFIDN, IFIDNI, IFNB, IFNDEF**

例

```
loadb MACRO source
  IFDIF <source>,<si>       ; 文字列ポインタを設定
    mov si,source
  ENDIF
    lodsb                   ; バイトを読み込む
  ENDM
```

IFE

機能　条件アセンブルブロックを開始します。式が偽 (0) の場合にブロックが有効になります。

モード　MASM, Ideal

構文　**IFE** *expression*

説明　*expression*は評価されて定数になる式で，前方参照のシンボル名を含んではなりません。式が0と評価された場合は，条件ブロック内の命令がアセンブルされます。条件アセンブルブロックを終了するには，**ENDIF**疑似命令を使ってください。

参照　**ELSE, ENDIF, IF, IF1, IF2, IFB, IFDEF, IFDIFI, IFIDN, IFIDNI, IFNB, IFNDEF**

例

```
IFE StackSize
StackSize = 1024
    DB  StackSize DUP (?)    ; スタックを確保
ENDIF
```

IFIDN, IFIDNI

機能 条件アセンブルブロックを開始します。2つの引数が同じ場合にブロックが有効になります。

モード MASM, Ideal

構文 **IFIDN** <*argument1*>,<*argument2*>

説明 **IFIDN**は常にマクロ内で使い，2つの引数が同じ文字列かどうかをチェックします。どちらかの引数としてマクロ命令のダミー（仮）引数を指定できます。マクロコールへのダミー（仮）引数は，比較実行前にマクロコールへの実引数に置き換えられます。2つの文字列が同じ場合は，条件ブロック内の命令がアセンブルされます。2つの文字列は，1文字ずつ大文字小文字の区別をつけて比較されます。大文字小文字の区別を無視したい場合は，**IFIDNI**疑似命令を使ってください。条件アセンブルブロックを終了するには，**ENDIF**疑似命令を使ってください。

参照 **ELSE, ENDIF, IF, IF1, IF2, IFB, IFDEF, IFDIF, IFDIFI, IFE, IFNB, IFNDEF**

例

```
rdwr MACRO buf,rwmode
    mov ax,buf
  IFIDN <rwmode>,<read>
    call ReadIt
  ENDIF
  IFIDN <rwmode>,<write>
    call WriteIt
  ENDIF
  ENDM
```

IFNB

機能　条件アセンブルブロックを開始します。引数がブランク以外の場合にブロックが有効になります。

モード　MASM, Ideal

構文　**IFNB** <*argument*>

説明　*argument*がブランク（空白）でない場合は，条件ブロック内の命令がアセンブルされます。**IFNB**疑似命令は，指定されたダミー(仮)引数と置き換わるための実引数を伴ってマクロが呼び出されたかどうかのチェックに使ってください。

チェックする引数は，常に不等号カッコ（<>）で囲んでください。

条件アセンブルブロックを終了するには，**ENDIF**疑似命令を使ってください。

参照　**ELSE, ENDIF, IF, IF1, IF2, IFB, IFDEF, IFDIF, IFDIFI, IFE, IFIDN, IFIDNI, IFNDEF**

例

```
PopRegs MACRO reg1,reg2
  IFNB <reg1>
    pop reg1
  ENDIF
  IFNB <reg2>
    pop reg2
  ENDIF
  ENDM
```

IFNDEF

機能 条件アセンブルブロックを開始します。シンボルが未定義の場合にブロックが有効になります。

モード MASM, Ideal

構文 **IFNDEF** *symbol*

説明 *symbol*が未定義の場合は，条件ブロック内の命令がアセンブルされます。
条件アセンブルブロックを終了するには，**ENDIF**疑似命令を使ってください。

参照 **ELSE, ENDIF, IF, IF1, IF2, IFB, IFDEF, IFDIF, IFDIFI, IFE, IFIDN, IFIDNI, IFNB**

例

```
IFNDEF BufSize
BufSize EQU 128      ; 未定義ならばバッファ(BufSize)を定義
ENDIF
```

%INCL

機能　インクルードファイルのリスト表示を有効にします。

モード　MASM, Ideal

構文　%INCL

説明　%INCLは，%NOINCL疑似命令でインクルードファイルのリスト表示を無効にした後で使ってください。特に指定のない限り，インクルードファイルのリスト表示は有効です。

参照　%NOINCL

例

```
%INCL
INCLUDE DEFS.INC      ; リスト表示される
```

INCLUDE

機能　他ファイルからソースコードをインクルードします（取り込みます）。

モード　MASM, Ideal

構文　MASMモード:

INCLUDE *filename*

Idealモード:

INCLUDE *"filename"*

説明　*filename*は，アセンブラ命令が入っているソースファイルの名前です。インクルードされたファイル内のすべての命令をアセンブルした後に，アセンブルが続行されます。

*filename*は通常のDOSのファイルの命名規定に従います。ドライブ名（オプション），ディレクトリ（オプション），ファイル名，拡張子（オプション）の順番で指定します。拡張子（拡張子）を省略すると，.ASMが指定されます。

*filename*にディレクトリまたはドライブ名を指定しない場合は，Turbo Dubuggerは/**I**コマンドラインオプションで指定されたディレクトリ内を最初に探し，次にカレントディレクトリ内を探します。

INCLUDEは，任意のレベルまでネストすることが可能です。

Idealモードでは，*filename*を引用符で囲む必要があります。

例

```
; MASMモード
INCLUDE MYMACS.INC      ; マクロ定義をインクルード
; Idealモード
INCLUDE "DEFS.INC"      ; イコール定義をインクルード
```

INCLUDELIB

機能　ライブラリをインクルードするようリンカに指示します。

モード　MASM, Ideal

構文　MASMモード:

INCLUDE *filename*

Idealモード:

INCLUDELIB *"filename"*

説明　*filename*はリンク時にインクルードしたいライブラリの名前です。拡張子を省略すると，.LIBが指定されます。

INCLUDELIB疑似命令は，ソースファイルが常に指定されたライブラリ内のルーチンを使うことがわかっている場合に使ってください。そうすれば，リンカコマンド内でライブラリ名を指定する必要はありません。

Idealモードでは，*filename*を引用符で囲んでください。

例

```
INCLUDELIB diskio     ; DISKIO.LIBをインクルード
```

INSTR

機能　文字列中の別の文字列の位置を返します。

モード　MASM51, Ideal

構文　*name* **INSTR** *start, string1, string2*

説明　*string1*内にある最初の*string2*の位置の値が*name*になります。*string1*の先頭文字の位置は1です。*string2*が*string1*内に存在しない場合は，0を返します。

参照　**CATSTR, SIZESTR, SUBSTR**

例

```
COMMAPOS INSTR <aaa,bbb>,<,>     ; COMMAPOS = 4
```

IRP

機能　文字列を置き換えながら命令ブロックを繰り返します。

モード　MASM, Ideal

構文　**IRP** *parameter,<arg1,arg2 ...>*

statements

ENDM

説明　繰り返しブロック内の*statement*が，カッコ内の引数リストの各引数について1回ずつアセンブルされます。リストには，任意の数の引数を入れることができます。引数は，シンボル，文字列，数値などの任意のテキストです。ブロックがアセンブルされるたびに，リスト内の次の引数が*parameter*に置き換わります。

引数リストは常に不等号カッコ（<>）で囲み，各引数はカンマで区切ります。

繰り返しブロックを終了するには，**ENDM**疑似命令を使ってください。

IRPは，マクロ命令の内側でも外側でも使うことができます。

参照　**IRPC, REPT**

例

```
IRP  reg,<ax,bx,cx,dx>
  push reg
ENDM
```

IRPC

機能　文字を置き換えながら命令ブロックを繰り返します。

モード　MASM, Ideal

構文　**IRP** *parameter, string*
　　statements
ENDM

説明　繰り返しブロック内の*statement*が，*string*中の各文字について1回ずつアセンブルされます。*string*文字列の文字数は任意です。ブロックがアセンブルされるたびに，文字列内の次の文字が*parameter*に置き換わります。
繰り返しブロックを終了するには，**ENDM**疑似命令を使ってください。
IRPCは，マクロ命令の内側でも外側でも使うことができます。

参照　**IRP, REPT**

例

```
IRPC lucky,1379
  DB   lucky         ; ラッキーナンバを定義
ENDM
```

この例は，1, 3, 7, 9の4バイトのデータを生成します。

JUMPS

機能　条件ジャンプ命令のnearまたはfarアドレスへの拡張を有効にします。

モード　MASM, Ideal

構文　**JUMPS**

説明　**JUMPS**疑似命令は，*short*変位で条件ジャンプ命令のデスティネーションアドレスまでジャンプさせることができない場合に，反対の意味の条件ジャンプ命令と通常のジャンプ命令のペアを生成します。たとえば，

```
jne  xyz
```

という条件ジャンプ命令を次のようなジャンプ命令のペアに置き換えます。

```
je  @@a
jmp xyz
a:
```

デスティネーションアドレスが前方参照の場合は，**NEAR**または**FAR**演算子を使って，ジャンプ命令のために確保する領域の大きさを指示する必要があります。そうしないと，適切なコードが生成されません。
この疑似命令には，コマンドラインオプションの/**JJUMPS**を使うのと同じ効果があります。

参照　**NOJUMPS**

例

```
JUMPS                   ; 条件ジャンプ命令の拡張を有効にする
  jne NEAR A            ; Aまで届く
  ORG $+400h
A:
```

LABEL

機能　指定された型のシンボルを定義します。

モード　MASM, Ideal

構文　MASMモード:

name **LABEL** *type*

Idealモード:

LABEL *name type*

説明　*name*は，ソースファイル内でまだ定義されていない新しいシンボルです。*type*は，シンボルのサイズとコードとデータのどちらを参照するのかを指定します。次のどれかです。

- **NEAR, FAR, PROC**のどれか。**PROC**は，**NEAR**または**FAR**です（**MODEL**疑似命令で設定されたメモリモデルによって異なります）。
- **BYTE, WORD, DWORD, FWORD, PWORD, QWORD, TBYTE,**構造体名，DATAPTRのどれか。

ラベルはカレントソースファイル内からのみアクセスすることができます。ただし，**PUBLIC**疑似命令を指定した場合は，他のソースファイルからもアクセスすることができます。

Idealモードでは，*name*は**LABEL**疑似命令の後に指定することに注意してください。

LABELは異なるサイズの項目をアクセスするために使用してください。次の例を参照してください。

参照　:

例

```
words LABEL WORD        ; "BYTES"をワード型として定義
BYTES DB 64 DUP (0)
    mov words[2],1      ; WORDに1を書き込む
```

.LALL

機能　マクロ命令の展開のリスト表示を有効にします。

モード　MASM

構文　**.LALL**

参照　**%MACS**

.LFCOND

機能　条件ブロック内のすべての命令をリスト表示します。

モード　MASM

構文　**.LFCOND**

説明　**.LFCOND**は，アセンブルリスト内の偽条件ブロックの表示を有効にします。**.LFCOND**は/**X**オプションによって影響されません。

参照　**%CONDS**

%LINUM

機能　リストファイルの行番号フィールドの幅を設定します。

モード　MASM, Ideal

構文　%LINUM *size*

説明　**%LINUM**は，リストファイルの行番号フィールドの桁数を設定します。*size*は桁数を指定する定数です。リスト幅をできるだけ狭くしたい場合は，このフィールド幅を小さく指定します。またソースファイル内に9999行を越える行がある場合は，このフィールド幅を大きくして行番号が切り捨てられないように指定します。

例

```
%LINUM 5    ; 5桁(99999行)までリスト表示する
```

%LIST

機能 ソース行のリスト表示を有効にします。

モード MASM, Ideal

構文 **%LIST**

説明 **%LIST**は，すべてのリスト表示を中断する**%NOLIST**と反対の処理を指定します。
これは，デフォルトのリストモードです。すべてのソース行がリスト表示されます。

参照 **.LIST, %NOLIST, .XLIST**

例

```
%LIST
    jmp xyz     ; この行は常にリスト表示される
```

.LIST

機能 リストにソース行を表示します。

モード MASM

構文 **.LIST**

参照 **LIST**

例

```
    .XLIST          ; リスト表示をオフにする
INCLUDE  more.inc
    .LIST           ; リスト表示をオンにする
```

LOCAL

機能　マクロまたはプロシージャのローカル変数を定義します。

モード　MASM, Ideal

構文　マクロ内:

LOCAL *symbol* [*,symbol*] ...

プロシージャ内:

LOCAL *name:type* [*:count*] [*,name:type* [*:count*]] ... = [*symbol*]

説明　**LOCAL**は, **MACRO**疑似命令で始まるマクロ定義内および**PROC**疑似命令で定義されるプロシージャ内で使うことができます。**LOCAL**の動作は, 使われる場所によって若干異なります。

マクロ定義内では, **LOCAL**は一時的な*symbol*の名前(マクロの展開のたびに新しい固有シンボルに置き換わります)を定義します。固有シンボルは, *??number*という形式を持ちます。*number*は0000～FFFFの範囲の16進数です。

プロシージャ内では, **LOCAL**はスタック領域にアクセスするシンボル（*name*）を定義します。スタックはBPレジスタの基づいた負のオフセットとしてアクセスされます。ローカル変数の先頭は, BP(*type*×*count*)から始まります。引数リストの最後に等号（=）とシンボル（*symbol*）を指定すると, そのシンボルはローカルシンボルブロックの合計サイズ（バイト数）を表わします。この値を使って, スタック領域内にローカル変数用の領域を確保することができます。

*localdef*の構文は次のとおりです。

```
localname:[ [distance] PTR]type[:count]
```

*localdef*に対して, 次の代替形式を使うことができます。

```
localname [ [count] ][:[[distance] PTR] type]
```

*localname*は, このローカルシンボルを参照するためにプロシージャ内で使う名前です。*type*は, 引数のデータ型で次のどれかです。

WORD, **DWORD**, **FWORD**, **PWORD**, **QWORD**, **TBYTE**, 構造体名, **DATAPTR**

代替形式を使った場合，*type*を省略すると**WORD**サイズが指定されます。
*count*は，スタック上に確保するtype型の要素の数を指定します。
オプションの*distance*と**PTR**は，Turbo Debugger用のデバッグ情報を入れることを指示します。同時に，このローカル変数が別のデータ型へのポインタであることを通知します。詳しくは，**PROC**疑似命令の説明を参照してください。
LOCALの使用例を次に示します。

```
LOCAL X:DOWRD:4,Y:NEAR PTR WORD
```

次に代替形式を使った**LOCAL**の使用例を示します。

```
LOCAL X[4]:DWORD,Y:PTR STRUCNAME
```

type（型）は，シンボルのために確保する領域サイズを示します。データの値に対しては**BYTE, WORD, DWORD, FWORD, PWORD, QWORD, TBYTE, DATAPTR**のどれかで，コードポインタに対しては**NEAR, FAR, PROC**のどれかです。
LOCAL疑似命令は，マクロ定義内ではその先頭に指定してください。プロシージャ内ではどこに置いてもかまいませんが，**LOCAL**で定義するシンボルは，**LOCAL**自身よりも前で使うことはできません。

参照　**ARG, MACRO, PROC, USES**

例

```
OnCarry MACRO func
    LOCAL done
    jnc  done           ; キャリがセットされてなければマクロ終わり
    call func           ; セットされていればファンクションコール
done:
    ENDM
read PROC NEAR
    LOCAL N:WORD = LSIZE
    push bp
    mov  bp,sp
    sub  sp,LSIZE       ; ローカル変数用に領域を確保
    mov  N,0            ; N = [BP-2]
    ; funcの本体
    add  sp,LSIZE       ; スタックを調整
    pop  bp
    ret
read ENDP
```

LOCALS

機能　ローカルシンボルの使用を有効にします。

モード　MASM, Ideal

構文　**LOCALS** [*prefix*]

説明　特に指定のない限り，ローカルシンボルの先頭には2つの単価記号（@@）を付けなければなりません。ローカルシンボルは，非ローカルシンボルで挟まれたブロック内でのみ参照することができます。非ローカルシンボルは，**PROC, LABEL**疑似命令またはコロン（：）演算子を使って定義することができます。**LOCALS**疑似命令を使うと，非ローカルシンボルで挟まれたシンボルはそのブロック内でのみ参照可能となります。ローカルシンボルの使用を有効にした場合は，プロシージャの他のブロック内で同じシンボルを再使用することができます。

*prefix*は2文字のシンボルプレフィクスで，ローカルシンボル名の先頭2文字を指定します。通常は，2つの単価記号（@@）がローカルシンボルを表わします。プログラムで2つの単価記号で開始するシンボルを使う場合，または独自の規則に基づいてローカルシンボルを使いたい場合は，ローカルシンボルの先頭2文字として単価記号以外の文字を指定することができます。先頭2文字としては，シンボル名の先頭に使える文字を指定できます。たとえば.?は使えますが，..は使えません。プレフィクスを設定すると自動的にローカルシンボルも有効となります。**NOLOCALS**疑似命令でローカルシンボルを無効にしても，プレフィクスは次に**LOCALS**疑似命令でローカルシンボルを有効にするまで記憶されています。

Idealモードではローカルシンボルは自動的に使用可能となります。**NOLOCALS**疑似命令を指定すれば，ローカルシンボルを使用不能にすることができます。その場合は，後続のすべてのシンボル名はソースファイル内のどこからでもアクセス可能になります。

参照　**NOLOCALS, IDEAL**

例

```
LOCAL
abc PROC
@@1:
    LOOP @@1
abc ENDP
xyz PROC
@@1:            ; ローカルシンボルとなり，他の@@1と二重定義にならない
    LOOP @@1
xyz ENDP
```

MACRO

機能　マクロ命令を定義します。

モード　MASM, Ideal

構文　MASMモード:

name **MACRO** [*parameter* [*,parameter*] ...]

Idealモード:

MACRO *name* [*parameter* [*,parameter*] ...]

説明　*name*は，マクロの名前です。後にソースファイル内でマクロを展開するときにこの名前を使うことができます。*parameter*は，マクロ定義のボディ内で使える仮引数です。実引数を伴うマクロ呼び出しの際に，実引数と置き換わります。

マクロ定義を終了するには，**ENDM**疑似命令を使ってください。

参照　**ENDM**

例

```
swap MACRO A,B      ; 2つのデータを入れ替えるマクロ
    mov ax,A
    mov A,B
    mov B,ax
    ENDM
```

%MACS

機能 マクロ展開のリスト表示を有効にします。

モード MASM, Ideal

構文 **%MACS**

説明 **%MACS**疑似命令は，直前の**%NOMACS**と反対の処理を指定します。マクロ展開の結果，生成された命令をリスト表示します（**.LALL**と同様です）。

参照 **.LALL, %NOMACS, .SALL, .XALL**

例

```
%MACS
mymac 1,2,3      ; マクロ展開後の命令をリスト表示する
```

MASM

機能　MASMのアセンブルモードにします。

モード　MASM, Ideal

構文　**MASM**

説明　**MASM**疑似命令は，MASMの大まかな式構文を受け付けるように式パーサに指示します。IdealモードとMASMモードの構文の違いについては，付録Bを参照してください。

Turbo Assemblerでは，最初のアセンブル開始時はMASMモードになっています。

参照　**QUIRKS, IDEAL**

例

```
MASM
mov al,es:24h        ; MASMのひどい構文
```

MASM51

機能 MASMのバージョン5.1の拡張機能のアセンブルを有効にします。

モード MASM, Ideal

構文 **MASM51**

説明 **MASM51**を指定すると，通常のTurbo Assemblerでは使うことのできない以下の機能が使用可能になります。

■SUBSTR, CATSTR, SIZESTR, INSTRの各疑似命令
■円記号（¥）による行継続機能（IBM PCではバックスラッシュ（＼）となります）

QUIRKS疑似命令でQuirksモードを有効にすると，さらに以下の機能が使用可能となります。

■＠＠で定義され，＠*F*と＠*B*で参照されるローカルラベル
■**PROC**内での変数の再定義
■C言語からコールされる先頭に下線記号(_)が付いた**PROC**は，すべて**PUBLIC**となる。

参照 **NOMASM51**

例

```
MASM51
MYSTR   CATSTR <ABC>,<XYZ>    ; MYSTR = "ABCXYZ"
```

.MODEL

機能　簡略セグメント疑似命令に対するメモリモデルを設定します。

モード　MASM

構文　**.MODEL** *memorymodel* [*,language*]
.MODEL TPASCAL

説明　*memorymodel*はメモリモデルで，タイニィ，スモール，ミディアム，コンパクト，ラージ，ヒュージのうちのどれかです。ラージとヒュージモデルでは，同じセグメント定義を使いますが，定義済みイコール定義シンボルの@*DataSize*の定義内容は異なります(@*DataSize*シンボルの説明は，**ユーザーズガイド**の第4章の「他の簡略セグメント疑似命令」を参照してください)。
Turbo Pascalとリンクするアセンブラモジュールを作る場合は，次のような特殊形式の**.MODEL**疑似命令を使ってください。

```
.MODEL TPASCAL
```

この疑似命令は，Turbo Pascalのセグメントの命名規定を使うように指示します。**TPASCAL**を指定した場合は，**.CODE**と**.DATA**簡略セグメント疑似命令のみを指定することができます。**.MODEL**疑似命令の第2引数を指定する必要はありません。**TPASCAL**がそれも兼ねています。Turbo Pascalのアセンブラモジュールで禁止されている疑似命令を使おうとすると，警告メッセージが表示されます。
*memorymodel*を定義するには，**.CODE, .DATA, .STACK**などの簡略セグメント疑似命令よりも先に**.MODEL**疑似命令を指定してください。**.MODEL**疑似命令より先に**.386**または**.386P**疑似命令を指定して80386プロセッサを有効にした場合は，コードセグメントとデータセグメントはともに32ビットセグメントとなります。この場合は，32ビットセグメントが本当に必要なことを確認してください。16ビットセグメントが必要な場合は，**.386**または**.386P**の前に**.MODEL**疑似命令を置いてください。
*language*は，モジュール内プロシージャのアクセスのためにモジュールがどの言語のプログラムからコールされるのかを指示します。*language*に指定できる言語は，C, Pascal, Basic, FORTRAN, Prologのどれかです。**PROC**と**ENDP**疑似

命令を使うと，プロシージャの入口と出口の適切なコードが自動生成されます。

MASM51疑似命令でMASMのバージョン5.1との互換を有効にして，*language*にC言語を指定し，なおかつ**QUIRKS**疑似命令を使用すると，すべてのパブリックシンボルと外部シンボルの先頭に下線記号(_)が付けられます。この理由は，Turbo Cは特に指定のない限り，すべてのシンボルの先頭に下線記号を付けるからです。

*language*はまた，呼び出し側モジュールがどの順番でプロシージャの引数をスタックにプッシュするのかを指示します。*language*にPascal, Basic, FORTRANのどれかを指定した場合は，呼び出し側ソースプログラム上の左端の引数を最初に（右へ向って）順番にスタックへプッシュされるものと判断します。*language*にCまたはPrologを指定した場合は，呼び出し側ソースプログラム上の右端の引数を最初に（左へ向って）順番にスタックへプッシュされるものと判断します。CまたはPrologを指定した場合はまた，呼び出し側の関数はプッシュした引数をスタックから削除することを前提とします。それ以外の言語を指定した場合は，呼び出し側へリターンする前にスタックの引数を削除するために適切な形式の**RET**命令が自動生成されます。

*language*を省略すると，**.MODEL**疑似命令は簡略セグメント疑似命令でセグメントがどのように使われるかを定義するだけです。

以下に各メモリモデルに対するデフォルトのセグメント属性を示します。

表3.1　タイニィメモリモデルのデフォルトのセグメントと型

疑似命令	名前	アラインメント	結合	クラス	グループ
.CODE	_TEXT	WORD	PUBLIC	'CODE'	DGROUP
.FARDATA	FAR_DATA	PARA	private	'FAR_DATA'	
.FARDATA?	FAR_BSS	PARA	private	'FAR_BSS'	
.DATA	_DATA	WORD	PUBLIC	'DATA'	DGROUP
.CONST	CONST	WORD	PUBLIC	'CONST'	DGROUP
.DATA?	_BSS	WORD	PUBLIC	'BSS'	DGROUP
.STACK	STACK	PARA	STACK	'STACK'	DGROUP

表3.2　スモールメモリモデルのデフォルトセグメントと型

疑似命令	名前	アラインメント	結合	クラス	グループ
.CODE	_TEXT	WORD	PUBLIC	'CODE'	
.FARDATA	FAR_DATA	PARA	private	'FAR_DATA'	
.FARDATA?	FAR_BSS	PARA	private	'FAR_BSS'	
.DATA	_DATA	WORD	PUBLIC	'DATA'	DGROUP
.CONST	CONST	WORD	PUBLIC	'CONST'	DGROUP
.DATA?	_BSS	WORD	PUBLIC	'BSS'	DGROUP
.STACK	STACK	PARA	STACK	'STACK'	DGROUP

表3.3　ミディアムメモリモデルのデフォルトのセグメントと型

疑似命令	名前	アラインメント	結合	クラス	グループ
.CODE	name_TEXT	WORD	PUBLIC	'CODE'	
.FARDATA	FAR_DATA	PARA	private	'FAR_DATA'	
.FARDATA?	FAR_BSS	PARA	private	'FAR_BSS'	
.DATA	_DATA	WORD	PUBLIC	'DATA'	DGROUP
.CONST	CONST	WORD	PUBLIC	'CONST'	DGROUP
.DATA?	_BSS	WORD	PUBLIC	'BSS'	DGROUP
.STACK	STACK	PARA	STACK	'STACK'	DGROUP

表3.4　コンパクトメモリモデルのデフォルトのセグメントと型

疑似命令	名前	アラインメント	結合	クラス	グループ
.CODE	_TEXT	WORD	PUBLIC	'CODE'	
.FARDATA	FAR_DATA	PARA	private	'FAR_DATA'	
.FARDATA?	FAR_BSS	PARA	private	'FAR_BSS'	
.DATA	_DATA	WORD	PUBLIC	'DATA'	DGROUP
.CONST	CONST	WORD	PUBLIC	'CONST'	DGROUP
.DATA?	_BSS	WORD	PUBLIC	'BSS'	DGROUP
.STACK	STACK	PARA	STACK	'STACK'	DGROUP

表3.5 ラージまたはヒュージメモリモデルのデフォルトのセグメントと型

疑似命令	名前	アラインメント	結合	クラス	グループ
.CODE	_TEXT	WORD	PUBLIC	'CODE'	
.FARDATA	FAR_DATA	PARA	private	'FAR_DATA'	
.FARDATA?	FAR_BSS	PARA	private	'FAR_BSS'	
.DATA	_DATA	WORD	PUBLIC	'DATA'	DGROUP
.CONST	CONST	WORD	PUBLIC	'CONST'	DGROUP
.DATA?	_BSS	WORD	PUBLIC	'BSS'	DGROUP
.STACK	STACK	PARA	STACK	'STACK'	DGROUP

表3.6 Turbo Pascal(TPASCAL)メモリモデルのデフォルトのセグメントと型

疑似命令	名前	アラインメント	結合
.CODE	CODE	BYTE	PUBLIC
.DATA	DATA	WORD	PUBLIC

参照　**.CODE, .DATA, .FARDATA, .FARDATA?, .STACK**

例

```
.MODEL MEDIUM     ; ミディアムモデル(スモールデータとラージコード)を指定
```

MODEL

機能　簡略セグメント疑似命令に対するメモリモデルを設定します。

モード　MASM, Ideal

構文　**MODEL**

参照　**.MODEL**

MULTERRS

機能　ソース行の1行に対して複数のエラー表示を有効にします。

モード　MASM, Ideal

構文　**MULTERRS**

説明　**MULTERRS**は，1行のソース行に対して複数のエラーメッセージまたは警告メッセージの表示を有効にします。見つかりにくいエラー原因を探したり，ソース行が複数のエラーを持つ場合に便利です。
最初のエラーに付随して，複数のエラーメッセージが発生（連鎖反応）することもありますので注意してください。これらの連鎖メッセージは，最初のエラーを修正すれば無くなることがあります。

参照　**NOMULTERRS**

例

```
MULTERRS
mov  ax,[bp+abc      ; 次の2種類のエラーメッセージが発生する
                     ; 1) Undefined symbol: abc (未定義シンボル: abc)
                     ; 2) Need right bracket (右大カッコが必要)
```

この例では，次の2つのエラーメッセージが表示されます。

```
**Error** MULTERRS.ASM(6) Undefined symbol: ABC
**Error** MULTERRS.ASM(6) Need right square bracket
```

NAME

機能　オブジェクトファイルモジュールの名前を設定します。

モード　MASM, Ideal

構文　**NAME** *modulename*

説明　この疑似命令は，**MASM**モードでは何もしません。Idealモードでのみ有効です。Turbo Assemblerでは，ソースファイル名に任意のドライブ，ディレクトリ，拡張子を付けてモジュール名とすることができます。**NAME**は，オブジェクトファイルのデフォルト名を変更したいときに使います。*modulename*がモジュールの新しい名前になります。

例

```
NAME loader
```

%NEWPAGE

機能　リストファイルを改ページします。

モード　MASM, Ideal

構文　**%NEWPAGE**

説明　**%NEWPAGE**の後のソース行が，リストファイルの次のページの先頭からリストされます（**PAGE**を引数なしで指定した場合も同様です）。

参照　**PAGE**

例

```
%NEWPAGE
; このコメント行は，次のページの先頭にリストされる
```

%NOCONDS

機能　偽条件ブロック内の命令のリスト表示を無効にします。

モード　MASM, Ideal

構文　**%NOCONDS**

説明　**%NOCONDS**の指定はリスト制御疑似命令の設定に優先します。リスト制御が有効になっている場合，条件ブロック内のすべての命令および偽と評価されたために無視されたブロック内のすべての命令はリスト表示されません(**.SFCOND**と同様です)。

参照　**%CONDS, .LFCOND, .SFCOND, .TFCOND**

例

```
%NOCONDS
IF 0
    mov ax,1      ; "IF 0"が指定されているのでリスト表示しない
ENDIF
```

%NOCREF

機能　クロスリファレンスリスト（CREF）の表示を無効にします。

モード　MASM, Ideal

構文　**%NOCREF** ［*symbol*］

説明　**%NOCREF**を指定すると，ソースファイルの以降のシンボル（*symbol*）のクロスリファレンス情報の蓄積処理を中止します。
引数を省略して**%NOCREF**だけを指定した場合は，クロスリファレンス処理そのものが中止します。1つまたは複数の*symbol*を指定した場合は，そのシンボルに対するクロスリファレンス処理を中止します（**.XCREF**と同様です）。

参照　**%CREF, .CREF, .XCREF, %CREFALL, %CREFREF, %CREFUREF**

例

```
%NOCREF xyz
wval DW 0        ; CREFによってWVALはクロスリファレンスリストに表示される
xyz  DB 0        ; XYZはクロスリファレンスリストに表示されない
```

%NOCTLS

機能　リスト制御疑似命令のリスト表示を無効にします。

モード　MASM, Ideal

構文　**%NOCTLS**

説明　**%NOCTLS**は，直前の**%CTLS**（リスト制御疑似命令をリストファイル内に入れることを指定する）と反対の指定を行ないます。**%NOCTLS**を指定すると，次の行からリスト制御疑似命令のリスト表示が無効となります。
%NOCTLSは，ソースアセンブル開始時のデフォルトのリスト制御モードです。

参照　**%CTLS**

例

```
%NOCTLS
%LIST     ; この疑似命令はリストファイルに入らない
```

NOEMUL

機能　エミュレーションコプロセッサ命令の生成を禁止します。

モード　MASM, Ideal

構文　**NOEMUL**

説明　**NOEMUL**は，80x87コプロセッサが実行する実浮動小数点命令を生成することを指示します。ファイルのある部分で実浮動小数点命令を生成し，他の部分でエミュレーション命令を生成したい場合は，**EMUL**と**NOEMUL**疑似命令を組み合わせて使います。

NOEMULは，通常の命令（実浮動小数点命令）のアセンブルモードを指定します。このモードは，ソースアセンブル開始時のデフォルトモードです。

参照　**EMUL**

例

```
NOEMUL          ; 実浮動小数点命令のアセンブルモード
finit
EMUL            ; エミュレーションモードに戻る
```

%NOINCL

機能　インクルードファイルのリスト表示を無効にします。

モード　MASM, Ideal

構文　**%NOINCL**

説明　**%NOINCL**を指定すると，以降のイクルードファイルのソース行のリスト表示を中止します。**%INCL**で再び有効になります。この疑似命令は，変更を加えないイコール定義などの多数のデータが入った大きなインクルードファイルに対して使うと便利です。

参照　**%INCL**

例

```
%NOINCL
INCLUDE DEFS.INC          ; リスト表示されない
```

NOJUMPS

機能　条件ジャンプ命令のジャンプ先の拡張を無効にします。

モード　MASM, Ideal

構文　**NOJUMPS**

説明　**NOJUMPS**疑似命令と**JUMPS**疑似命令を組み合わせて使うと，ソースファイル内の任意の場所でデスティネーションアドレスまでジャンプさせるための条件ジャンプ命令の拡張を行なったり中止したりすることができます。
このモードは，ソースアセンブル開始時のデフォルトモードです。

参照　**JUMPS**

%NOLIST

機能　ソース行のリスト表示を無効にします。

モード　MASM, Ideal

構文　**%NOLIST**

説明　**%NOLIST**は，すべてのリスト表示を中止します。**%LIST**を指定すれば，リスト表示が再開します。この疑似命令の指定は，他のすべてのリスト制御疑似命令の設定に優先します。

参照　**%LIST, .LIST, .XLIST**

例

```
%NOLIST
    add dx,ByteVar          ; この行はリスト表示されない
```

NOLOCALS

機能　ローカルシンボルの使用を無効にします。

モード　MASM, Ideal

構文　**NOLOCALS**

説明　**LOCALS**疑似命令によってローカルシンボルの使用が有効になっている場合は,2つの単価記号（@@）で始まるシンボルはローカルシンボルと見なされます。先頭に2つの単価記号が付くシンボルをローカルシンボルとして使いたくない場合は，この疑似命令を適当な場所で指定してください。
MASMモードでは,特に指定のない限りローカルシンボルの使用は無効となっています。

参照　**LOCALS, MASM**

例

```
NOLOCALS
@@buf DB 10 DUP (?)     ; 非ローカルシンボルの先頭に@@が使える
```

%NOMACS

機能　コードを生成しないマクロ展開命令のリスト表示を無効にします。

モード　MASM, Ideal

構文　%NOMACS

説明　**%NOMACS**疑似命令は，命令コードを生成しないソース行（コメント，**EQU**定義, **=**定義, **SEGMENT**疑似命令, **GROUP**疑似命令など)のリスト表示を無効にします。
このモードは，ソースアセンブル開始時のデフォルトのマクロリスト表示モードです（**.XALL**と同様です)。

参照　**.LALL, %MACS, .SALL**

NOMASM51

機能　MASMのバージョン5.1の拡張機能のアセンブルを無効にします。

モード　MASM, Ideal

構文　**NOMASM51**

説明　**NOMASM51**を指定すると，**MASM51**疑似命令で説明したMASM5.1の拡張機能が無効になります。このモードは，ソースアセンブル開始時のデフォルトモードです。

参照　**MASM51**

例

```
MASM51
SLEN SIZESTR <ax,bx>   ; SLEN = 5
NOMASM51
CATSTR PROC NEAR       ; MASM5.1モードが無効なのでCATSTRを
                       ; ユーザシンボルとして使うことが可能
;
CATSTR ENDP
```

NOMULTERRS

機能 ソース行の1行に対して1つのエラー表示を行ないます。

モード MASM, Ideal

構文 **NOMULTERRS**

説明 **NOMULTERRS**を指定すると,1行のソース行に対して1つのエラーメッセージまたは警告メッセージの表示を行ないます。1行に複数のエラーがある場合は,Turbo Assemblerは重要なエラーを優先して表示します。多くの場合,主要なエラーを訂正すれば他のエラーも無くなります。エラーメッセージの重要度を自分で判断したい場合は,**MULTERRS**疑似命令を使って1行に複数のエラー表示を指定してください。

このモードは,ソースアセンブル開始時のデフォルトエラー表示モードです。

参照 **MULTERRS**

例

```
NOMULTERRS
mov   ax,[bp+abc        ; エラーメッセージは次の1種類だけ表示される
                        ; 1) Undefined symbol: abc (未定義シンボル: abc)
```

この例では,次のエラーメッセージだけが表示されます。

```
**Error** MULTERRS.ASM(6) Undefined symbol: ABC
```

%NOSYMS

機能　シンボルテーブルのリスト表示を無効にします。

モード　MASM, Ideal

構文　%NOSYMS

説明　%NOSYMSを指定すると，シンボルテーブルはリストファイルに出力されません。通常は，リストファイルの最後にシンボルテーブル（ソースファイル内で定義されたすべてのシンボルの一覧）が出力されます。

参照　%SYMS

例

```
%NOSYMS      ; シンボルテーブルは作られない
```

%NOTRUNC

機能 リストファイル内の長すぎるフィールドをワードラップします（次行へ送ります）。

モード MASM, Ideal

構文 **%NOTRUNC**

説明 リストファイルのオブジェクトコードのフィールドは，ほとんどの命令コードの生成とデータ定義に対して十分な幅を持っています。このフィールドの幅は，**%BIN**疑似命令で調整することができます。1行に入りきらないコードを生成する場合は，通常はあふれた部分は切り捨てられるのでリスト上では見ることはできません。生成されたコードのすべてをリスト上で見たい場合は，この**%NOTRUNC**疑似命令を指定してください。

%NOTRUNCはまた，リストファイルの1行に入りきらないソース行を切り捨てるか次行へ送るかを指定することができます。

ソースフィールドの幅を設定するには，**%TEXT**疑似命令を使います。

参照 **%BIN, %TEXT, %TRUNC**

例

```
%NOTRUNC
    DQ 4 DUP (1.2,3.4)     ; 複数行に分割してリストする
```

NOWARN

機能　警告メッセージの表示を無効にします。

モード　MASM, Ideal

構文　**NOWARN** [*warnclass*]

説明　*warnclass*を付けないで**NOWARN**を指定すると，すべての警告メッセージの表示は無効になります。警告メッセージの識別シンボルを付けて**NOWARN**を指定すると，そのメッセージだけが無効になります。すべての警告メッセージには3文字の識別シンボルが割り当てられており，**WARN**疑似命令で使うことができます。これらのシンボルは，/**W**コマンドラインオプションでも指定することができます。

参照　**WARN**

例

```
NOWARN   OVF          ; 算術オーバフローメッセージを無効にする
DW 1000h * 1234h      ; メッセージは表示されない
```

ORG

機能　カレントセグメントのロケーションカウンタを設定します。

モード　MASM, Ideal

構文　**ORG** *expression*

説明　*expression*に前方参照のシンボル名を入れることはできません。*expression*には定数，カレントセグメント内のシンボルからのオフセット，$（カレントロケーションカウンタ）からのオフセットのどれかを指定することができます。

セグメントの中にすでにストア（格納）したデータまたはコードの前の位置まで，ロケーションカウンタの値を戻すことが可能です。これによって，テーブルの定義時には未定であったテーブル項目を後で入れることができます。ただしこの技法を使う際は，誤って何らかのデータを上書きする可能性があるので注意を要します。

ORG疑似命令を使って，ラベルを特定の絶対アドレスに結びつけることができます。また，.COMファイルの開始アドレス（**ORG** 100h）を設定することもできます。

参照　**SEGMENT**

例

```
PROG SEGMENT
    ORG 100h          ; .COMファイルの開始オフセット
```

%OUT

機能　画面にメッセージを表示します。

モード　MASM

構文　**%OUT** *text*

説明　*text*は，画面に表示させたいメッセージです。メッセージには単純な文字をだけを入れることができます。メッセージは標準出力デバイス（通常は画面）に出力されます。必要に応じて，DOSのリダイレクト機能を使って出力デバイスを画面からファイルへ切り替えることもできます。

この**%OUT**疑似命令をソースコードの各所に入れておけば，現在アセンブルされている箇所を知ることができます（**DISPLAY**と同様です）。

次のようにメッセージ内に代用文字列（&）を入れることもできます。

```
MAKE_DATA MACRO VALUE
          %OUT Initializing a byte to: &VALUE&
          DB   VALUE
          ENDM
```

参照　**DISPLAY**

例

```
%OUT  グラフィックドライバをアセンブルしています.
```

P186

機能 80186命令のアセンブルを有効にします。

モード MASM, Ideal

参照 **.186, .8086, .286, .286C, .286P, .386, .386C, .386P, P8086, P286, P286P, P386, P386P**

P286

機能 80286のすべての命令のアセンブルを有効にします。

モード MASM, Ideal

参照 **.8086, .186, .286C, .286P, .286, .386, .386C, .386P, P8086, P286P, P386, P386P**

P286N

機能 80286の非特権命令のアセンブルを有効にします。

モード MASM, Ideal

参照 **.8086, .186, .286C, .286P, .286, .386, .386C, .386P, P8086, P286, P286P, P386, P386P**

P287

機能 80287数値演算コプロセッサの命令のアセンブルを有効にします。

モード MASM, Ideal

参照 **.8087, .287, .387, P8087, PNO87, P387**

P386

機能 80386のすべての命令のアセンブルを有効にします。

モード MASM, Ideal

参照 **.8086, .186, .286C, .286, .286P, .386C, .386P, .386, P8086, P286, P286P, P386P**

P386N

機能 80386の非特権命令のアセンブルを有効にします。

モード MASM, Ideal

参照 **.8086, .186, .286C, .286, .286P, .386C, .386P, .386, P8086, P286P, P386, P386P**

P387

機能 80387数値演算コプロセッサの命令のアセンブルを有効にします。

モード MASM, Ideal

参照 **.8087, .287, .387, P8087, PNO87, P287**

P8086

機能 8086の命令のみのアセンブルを有効にします。

モード MASM, Ideal

参照 **.186, .286C, .286, .286P, .386C, .386, .386P, 8086, P286, P286P, P386, P386P**

P8087

機能 8087の命令のみのアセンブルを有効にします。

モード MASM, Ideal

参照 **.287, .387, .8087, PNO87, P287, P387**

PAGE

機能　リストページの行数と桁数を設定します。または改ページします。

モード　MASM

構文　**PAGE** [*rows*] , [*cols*]
PAGE +

説明　*rows*は1ページの行数（10～255）を指定します。*cols*は1行の桁数（59～255）を指定します。*rows*と*cols*を省略すると，パラメータの現在の設定はそのままです。1行の桁数だけを変更する場合は，桁数の前にカンマ（,）を付けてください。カンマを付けないと，1ページの行数を変更してしまいます。

PAGE疑似命令の後にプラス符号（+）を付けると，改ページします（セクション番号がインクリメントされ，ページ番号が1に戻ります）。

何も引数を付けないで**PAGE**疑似命令を指定すると，改ページしますがセクション番号は変化しません。

参照　**%NEWPAGE, %PAGESIZE**

例

```
PAGE        ; 改ページ
PAGE ,80    ; 桁数を80に設定，行数は変わらない
```

%PAGESIZE

機能　リストのページの行数と桁数を設定します。

モード　MASM, Ideal

構文　**%PAGESIZE** [*rows*], [*cols*]

説明　*rows*は1ページの行数（10～255）を指定します。*cols*は1行の桁数（59～255）を指定します。

*rows*と*cols*を省略すると，パラメータの現在の設定はそのままです。1行の桁数だけを変更する場合は，桁数の前にカンマ（,）を付けてください。カンマを付けないと，1ページの行数を変更してしまいます。

参照　**PAGE**

例

```
%PAGESIZE 66,132      ; ページ幅を大きめに，ページ高を普通に設定
%PAGESIZE ,80         ; 行数はそのまま，桁数を80に設定
```

%PCNT

機能　リストファイルのセグメント オフセットフィールドの幅を設定します。

モード　MASM, Ideal

構文　**%PCNT** *width*

説明　*width*は，アセンブルされているカレントセグメント内のオフセットフィールドに対して確保したい桁数です。特に指定のない限り，桁数は4(通常の16ビットセグメント）または8（386プロセッサで使われる32ビットセグメント）に設定されています。**%PCNT**による設定は，これらの桁数のデフォルト値に優先します。

参照　**%BIN, %DEPTH, %LINUM**

例

```
%PCNT 3
ORG 1234h     ; 234だけがリスト表示される
```

PNO87

機能　コプロセッサ命令のアセンブルを禁止します。

モード　MASM, Ideal

構文　**PNO87**

説明　Turbo Assemblerでは，通常は80x87コプロセッサファミリの命令をアセンブルすることができます。**PNO87**疑似命令を使えば，誤ったコプロセッサの命令の使用を防ぐことができます。またソフトウェアに浮動小数点エミュレーションパッケージが組み込まれていない場合にこの疑似命令を指定しておけば，数値演算コプロセッサのないシステムでプログラムを実行したことを認識することができます。

参照　**.8087, .287, .387, P8087, P287, P387**

例

```
PNO87
Fadd        ; エラーが発生する
```

%POPLCTL

機能　スタックからリスト制御情報をポップ（復帰）します。

モード　MASM, Ideal

構文　%POPLCTL

説明　**%POPLCTL**は，リスト制御の設定を直前の**%PUSHLCTL**の発行時点の状態に戻します。有効/無効を切り替えることができるリスト制御疑似命令（**%MACS, %LIST, %INCL**など）の設定が元に戻ります。フィールド幅を設定するリスト制御疑似命令（**%DEPTH, %PCNT**など）の設定は元に戻りません。これらのリスト制御の設定は，**%PUSHLCTL**で16レベルのスタックに退避することができます。**%POPLCTL**疑似命令をマクロまたはインクルードファイル内で指定すると，特殊なリスティングモード（マクロやインクルードファイルの展開が終わると無効となる）を呼び出して使うことができます。

参照　%PUSHLCTL

例

```
%PUSHLCTL
%NOLIST
%NOMACS
    .
    .
    .
%POPLCTL        ; リスト制御の設定を復帰
```

PROC

機能　プロシージャの開始を定義します。

モード　MASM, Ideal

構文　MASMモード:

name **PROC** [*distance*] [**USES** *registers,*]
[*argument* [**RETURNS** *argument* [*,argument*] ...] [*,argument*] ...]

Idealモード:

PROC *name* [*distance*] [**USES** *registers,*]
[*argument* [**RETURNS** *argment* [*, argument*] ...] [*,argument*] ...]

説明　*name*はプロシージャの名前です。オプションの*distance*は**NEAR**または**FAR**のどちらかで，コードのメモリモデルのデフォルトサイズに合わされます（**.MODEL**疑似命令などの）簡略セグメント疑似命令を使わない場合は，デフォルトサイズは**NEAR**になります。タイニィ，スモール，コンパクトの各メモリモデルでもデフォルトサイズはやはり**NEAR**です。その他のメモリモデルでは**FAR**となります。*distance*によって，プロシージャ内の**RET**命令からnearまたはfarのどちらのリターンコードを生成するかが決まります。原則として，**FAR**プロシージャは**FAR**型（セグメントとオフセットのペア）**CALL**命令によってコールされ，**NEAR**プロシージャは**NEAR**。（オフセットのみ）**CALL**命令によってコールされます。

*register*はプロシージャが使うレジスタリストです。*register*はプロシージャの入口でプッシュし，出口でポップしてください。レジスタ名をスペースで区切れば，複数のレジスタを指定することができます。

*argument*は，プロシージャコールの際に引き渡される引数を示します。**.MODEL**疑似命令で指定された言語は，引数がスタック上で逆順になるかどうかをチェックします。プロシージャにおける引数の順番は，プロシージャをコールする高級言語の関数における順番と同じにしてください。Turbo Assemblerは，必要に応じて引数を逆順に読みます。*argument*は次の構文形式を持ちます。

```
argname[:[[distance] PTR:] type]
```

*argname*は，プロシージャ内で引数を利用するときに使う引数の名前です。*distance*

はオプションで，引数が指定サイズ（**NEAR**または**FAR**）のポインタであることを示します。*type*は引数のデータ型で，**WORD, DWORD, FWORD, PWORD, QWORD, TBYTE,** 構造体名，DATAPTRのどれかです。

*type*を省略すると**WORD**が指定されます。

PTRを付けて引数がデータへのポインタであることを示すと，Turbo Debugger用のデバッグ情報を生成します。つまり，**PTR**を使用してもTurbo Assemblerが生成するコードには影響を与えませんが，追加デバッグ情報の生成に影響を与えます。実際のデータをアクセスするためには，ポインタ引数を使ってコードを書いてください。

PTRの前の**NEAR**または**FAR**を省略すると，カレントメモリモデルに基づいてポインタサイズが決まります。ただし386プロセッサについては，カレントセグメントアドレスサイズ（16または32ビット）に基づいてポインタサイズが決められます。ポインタサイズは，タイニィ，スモール，ミディアムの各メモリモデルでは**WORD**に，その他の16ビットセグメントを使うメモリモデルでは**DWORD**になります。386を使って32ビットセグメントをアクセスする場合は，ポインタサイズは，タイニィ，スモール，ミディアムの各メモリモデルでは**DWORD**に，コンパクト，ラージ，ヒュージの各メモリモデルでは**FWORD**になります。

オプションの**RETURNS**キーワードには，1つまたは複数の引数を指定します。これらの引数は，プロシージャが呼び出し側プログラムへリターンする際にスタックからポップされません。通常，**.MODEL**疑似命令で**TPASCAL**のPascal言語を指定すると，プロシージャからのリターン時にすべての引数がポップされます。**RETURNS**キーワードの後に引数を付けると，呼び出し側プログラムが利用できるように引数はスタック上に残されます。特にPascalの文字列のリターン値を定義するには，**RETURNS**キーワードの後にそれを指定してください。

プロシージャ定義を終了させるには，**ENDP**疑似命令を使います。プロシージャから呼び出し側へ戻るためには，**ENDP**の前に**RET**命令を入れなければなりません。

PROC/ENDPブロック内では，プロシージャの外からは参照できないローカルシンボルを使うこともできます。ローカルシンボルの先頭には2つの単価記号（@@）を付けます。

必要ならば**PROC/ENDP**疑似命令をネストすることが可能です。その場合は，ローカルシンボルもネストされます。

参照　**ARG, ENDP, LOCAL, LOCALS, USES**

例

```
READLINE proc near
    ; プロシージャ本体
READLINE endp

call READLINE
```

PUBLIC

機能　他のモジュールからアクセス可能なシンボルを宣言します。

モード　MASM, Ideal

構文　**PUBLIC** *symbol* [,*symbol* ...]

説明　*symbol*で指定されたシンボルは，他のモジュールから参照することができるようにオブジェクトファイル内に生成されます。これらのシンボルは，他のモジュールから参照することができます。シンボルをパブリックにしない場合は，シンボルはカレントソースファイルからのみアクセスできます。
パブリックシンボルとして宣言できるシンボルを次に示します。

- データ変数の名前
- プログラムラベル
- EQUで定義した数値定数

参照　**COMM, EXTRN, GLOBAL**

例

```
PUBLIC xyproc     ; プロジージャをパブリック宣言する
xyproc PROC NEAR
```

PURGE

機能　マクロ定義を取り除きます。

モード　MASM, Ideal

構文　**PURGE** *macroname* [,*macroname* ...]

説明　**PURGE**は*macroname*で指定されたマクロ定義を取り除きます。*macroname*をカンマで区切れば，複数のマクロ定義を取り除くことができます。
この疑似命令は，既存の機械命令または疑似命令の名前と同一名を使って定義したマクロを取り消して，それらの命令の機能を本来の機能に戻す場合に使います。

参照　**MACRO**

例

```
PURGE add
    add ax,4        ; 通常のadd命令として機能する
```

%PUSHLCTL

機能　スタックへリスト制御情報をプッシュ（退避）します。

モード　MASM, Ideal

構文　**%PUSHLCTL**

説明　**%PUSHLCTL**は，現在のリスト制御の設定を16レベルのスタックにプッシュ（退避）します。有効/無効を切り替えることができるリスト制御疑似命令（**%INCL**や**%NOINCL**など）の設定が退避されます。フィールド幅を設定するリスト制御疑似命令の設定は退避されません。**%PUSHLCTL**疑似命令をマクロまたはインクルードファイル内で指定すると，特殊なリスティングモード（マクロやインクルードファイルの展開が終わると無効となる）を退避して後で呼び出すことができます。

参照　**%POPLCTL**

例

```
%PUSHLCTL          ; リスト制御の設定を退避
%NOINCL
%MACS
%POPLCTL           ; リスト制御の設定を復帰
```

QUIRKS

機能　MASMのバグの利用を有効にします。

モード　MASM, Ideal

構文　**QUIRKS**

説明　**QUIRKS**を指定すると，特定のMASMのバグを利用するソースファイルをアセンブルすることができます。偶然にしか動作しないようなソースコードの存在を許すだけなので，できるだけこの疑似命令は使わないようにしてください。むしろソースコードを訂正して正常に動作するようにすべきです。
Quirksモードの詳細については，付録Bの「Turbo AssemblerのQuirksモード」の節を参照してください。

参照　**MASM, IDEAL**

例

```
QUIRKS
bval DB 0
mov  bval,es     ; レジスタの内容を1バイトアドレスへ転送
```

.RADIX

機能　整数定数に対するデフォルトの基数を設定します。

モード　MASM

構文　**.RADIX** *expression*

説明　*expression*は，評価されて2, 8, 10, 16のどれかになります。*expression*の中で指定する定数は，現在の基数の設定に関係なく常に10進数と見なされます。

例

```
.RADIX 8        ; 8進数にセット
  db 77         ; 10進数の63を定義
```

RADIX

機能　整数定数に対するデフォルトの基数を設定します。

モード　MASM, Ideal

構文　**RADIX**

参照　**.RADIX**

RECORD

機能　ビットフィールドから構成されるレコードを定義します。

モード　MASM, Ideal

構文　MASMモード:

name **RECORD** *field* [,*field* ...]

Idealモード:

RECORD *name field* [,*field* ...]

説明　*name*はレコードの名前です。このレコードフォーマットでメモリを確保するときはこの名前を使います。*field*は，次の形式でレコード内のビットグループを記述します。

```
fieldname:width[=expression]
```

*fieldname*は，レコード内のフィールド名です。式中に*fieldname*を使うと，その値はフィールドの最下位ビットを（レコードの）構成バイトまたは構成ワードの最下位ビットに一致させるために右へシフトすべきビット数を表わします。

*width*は1～16の範囲の定数で，フィールドのビット数を指定します。レコード内の全フィールドの合計ビット数は，32以下でなくてはなりません。合計ビット数が8以下ならばレコードサイズは1バイト，9～16ならばレコードサイズは2バイト，17～32ならばレコードサイズは4バイトとなります。

*expression*はフィールドのデフォルト値を指定するオプションのフィールドです。直前に等号(=)を付ける必要があります。*name*を使ってメモリ領域を定義した場合は，特に指定のない限りこのデフォルト値がフィールドに適用されます。レコードの上位にある未使用ビットは0に初期化されます。

RECORD疑似命令では，最初に定義されたフィールドがレコードの最上位ビットに，2番目に定義されたフィールドがそのすぐ下位のビットに，という順番で割り当てられます。全フィールドの合計ビット数が8または16にならない場合は全フィールドは右へシフトされるので，最後のフィールドの最下位ビットと（レコードの）構成バイトまたは構成ワードの最下位ビットは一致します。

参照　**STRUC**

例

```
myrec RECORD val:3=4, mode:2, size:4
```

REPT

機能　命令ブロックを繰り返します。

モード　MASM, Ideal

構文　**REPT** *expression*

statements

ENDM

説明　*expression*は，評価されて定数になります。前方参照シンボル名を含んではなりません。

繰り返しブロック内の*statement*が式で指定された回数だけアセンブルされます。

REPTは，マクロ命令の内側でも外側でも使うことができます。

参照　**ENDM, IRP, IRPC**

例

```
REPT 4
  shl ax,1
ENDM
```

.SALL

機能　マクロ展開でのすべての命令のリスト表示を無効にします。

モード　MASM

構文　**.SALL**

説明　マクロ命令の展開内容が見たいときは，**.SALL**を使ってリストファイルのサイズを減らしてください。

参照　**.LALL, .XALL, %MACS, %NOMACS**

例

```
.SALL
    mymacro 4      ; マクロを呼び出す
    add ax,si      ; リストではこの命令がMYMACROの次にくる
```

SEGMENT

機能　すべての属性を指定して，セグメントを定義します。

モード　MASM, Ideal

構文　MASMモード:

name **SEGMENT** [*align*] [*combine*] [*use*] [*'class'*]

Idealモード:

SEGMENT *name* [*align*] [*combine*] [*use*] [*'class'*]

説明　*name*はセグメントの名前です。同じ名前のセグメントがすでに定義されている場合は，このセグメントは以前定義されたセグメントの続きとして扱われます。
別のソースファイル内で同じセグメント名を使うことができます。リンカは同一名のセグメントを1つのセグメントとして結合 (bind) して実行可能プログラムを作ります。
*align*は，セグメントを開始するメモリ境界（アラインメント）の型を指定します。型は次のどれかを指定できます。

BYTE	次のバイトアドレスからセグメントを開始します。
WORD	次のワード（偶数）アドレスからセグメントを開始します。
DWORD	次のダブルワード（4の倍数）アドレスからセグメントを開始します。
PARA	次のパラグラフ（16の倍数）アドレスからセグメントを開始します。
PAGE	次のページ（256の倍数）アドレスからセグメントを開始します。

アラインメントの型を省略すると，**PARA**がデフォルトのアラインメント型として指定されます。
*combine*は，リンク時に別モジュールの同一名セグメント同士を結合する際の方法を指定します。以下の方法のどれかを指定できます。

■**AT** *expression*：

*expression*で指定された絶対パラグラフ（16の倍数）アドレスからセグメントを配置します。*expression*内では前方参照のシンボル名を使うことはできません。

ATセグメントに対して，リンカはデータまたはコードのどちらも生成しません。通常は，画面データ領域やROMのBIOSデータ領域などの固定メモリアドレスに対して，シンボル名を使ってアクセスする際にATセグメントを使います。

■**COMMON:**

このセグメントと他のすべての同一名セグメントを同じアドレスから配置します。結果として作られる共通セグメントのサイズは，同一名セグメントのうちで最大のセグメントのサイズと一致します。

■**MEMORY:**

すべての同一名セグメントを連結して1つの連続セグメントにします。**PUBLIC**結合型と同様です。

■**PRIVATE:**

このセグメントはどのセグメントとも結合されません。

■**PUBLIC:**

すべての同一名セグメントを連結して1つの連続セグメントにします。セグメントの合計サイズは，同一名セグメントのそれぞれのサイズの和となります。

■**STACK:**

すべての同一名セグメントを連結して1つの連続セグメントにします。リンカはスタックセグメント（SS）レジスタをこのセグメントの先頭に，スタックポインタ（SP）レジスタをこのセグメントのサイズに，それぞれ初期化します。これによって，プログラムはSSとSPをセットしないで，この結合型のセグメントをスタックの呼び出しとして使うことが可能となります。セグメントの合計サイズは，同一名セグメントのサイズの和となります。

結合型を省略すると，**PRIVATE**がデフォルトの結合型として指定されます。

*use*は，セグメントのデフォルトワードサイズを指定します。これは，**P386**または**P386N**疑似命令で80386プロセッサを有効にしないと使うことはできません。次のどちらかを指定できます。

■**USE16:**

セグメント属性のuseを省略した際に指定されるデフォルトのセグメント型です。**USE16**型のセグメントには，最大64Kバイトのコードまたはデータ（の一方または両方）を入れることができます。この型のセグメント内で32ビットのセグメント，レジスタ，定数のどれかを参照する場合は，デフォルトの16ビットサイズをオーバライドするための命令コードが新たに生成されます。

■**USE32:**

USE32型のセグメントには，最大4G（ギガ）バイトのコードまたはデータ（の一方または両方）を入れることができます。この型のセグメント内で16ビットのセグメント，レジスタ，定数のどれかを参照する場合は，デフォルトの32ビットサイズをオーバライドするための命令コードが新たに生成されます。

*class*は，リンク時のセグメントの順序を指定します。同じクラス名を持つセグメントは，ソースファイル内での順序に関係なく，同時にメモリにロードされます。クラス名は引用符（'または"）で囲んでください。

SEGMENT疑似命令で開始したセグメントを終了する場合は，**ENDS**疑似命令を指定します。Turbo Assemblerでは，セグメント疑似命令をネストすることはできますが，セグメントをネストすることはできません。**ENDS**疑似命令でネストセグメントを終了すると，以後データまたはコードはオリジナルのセグメントに追加されるだけです。

参照　**GROUP, MODEL, CODESEG, DATASEG**

例

```
PROG SEGMENT PARA PUBLIC 'CODE'
      .
      .
      .
PROG ENDS
```

.SEQ

機能　シーケンシャルセグメントオーダを指定します。

モード　MASM

構文　**.SEQ**

説明　**.SEQ**を指定すると，セグメントの順序付けはソースファイル内における順序と同じになります。ソースファイルのアセンブル開始時にはこのセグメントオーダが指定されています。**DOSSEG**疑似命令は，セグメントの順序に影響を与えます。**.SEQ**の機能は，/**S**コマンドラインオプションと同じ機能を持ちます。/**A**コマンドラインオプションでアルファベットセグメントオーダが指定されている場合は，**.SEQ**の指定が優先します。

参照　**.ALPHA, DOSSEG**

例

```
.SEQ
xyz SEGMENT     ; このセグメントが先頭になる
xyz ENDS
abc SEGMENT
abc ENDS
```

.SFCOND

機能　偽条件ブロック内の命令のリスト表示を禁止します。

モード　MASM

構文　**.SFCOND**

説明　**%CONDS, .LFCOND, %NOCONDS, .TSCOND**

SIZESTR

機能　文字列内の文字数を返します。

モード　MASM51, Ideal

構文　*name* **SIZESTR** *string*

説明　*name*に文字列内の文字数が返されます。ヌル文字列<>の長さは0です。

参照　**SUBSTR, CATSTR, INSTR**

例

```
REGLIST EQU <si di>
REGLEN SIZESTR REGLIST     ; REGLEN = 5
```

.STACK

機能　スタックセグメントの開始を定義します。

モード　MASM

構文　.STACK ［*size*］

説明　*size*はスタックのサイズをバイト数で指定します。サイズを省略すると，1Kバイト（1024バイト）となります。
.STACKは，通常スタンドアロン（独立した）プログラムを作る場合にのみ必要となります。高級言語からコールされるアセンブラルーチンを作る場合は，通常は高級言語側で必要なスタックを確保します。

参照　**.CODE, .CONST, .DATA, .DATA?, .FARDATA, .FARDATA?, .MODEL, STACK**

例

```
.STACK  200h        ; 512バイトのスタックを確保
```

STACK

機能　スタックセグメントの開始を定義します。

モード　MASM, Ideal

参照　**.CODE, .CONST, .DATA, .DATA?, .FARDATA, .FARDATA?, .MODEL, .STACK**

STRUC

機能　構造体を定義します。

モード　MASM, Ideal

構文　MASMモード:

name **STRUC**
　fields
[*name*] **ENDS**

Idealモード:

STRUC *name*
　fields
ENDS [*name*]

説明　IdealモードとMASMモードにおける**STRUC**疑似命令の違いは,1行目の**STRUC**と構造体名の順序,および2行目の**ENDS**とオプションの構造体名の順序が反対であることです。**STRUC**疑似命令をネストしたり，**UNION**疑似命令と**STRUC**疑似命令を組み合わせて使うことができます。

*name*は構造体の名前（タグ名）です。後でこのフォーマットの構造体としてメモリを確保する際に，この名前を使うことができます。

*fields*は構造体の構成フィールドを定義します。それぞれのフィールドのサイズは，通常のデータ定義疑似命令（**DB, DW**など）を使って定義します。*fields*には，名前を指定しても指定しなくてもかまいません。フィールド名は，シンボル名と同様にプログラム内で固有でなくてはなりません。ただしIdealモードでは固有である必要はありません。

フィールド定義において，データ定義疑似命令の後にデフォルト値を指定することができます。これは各データを初期化することと同じです。デフォルト値を指定しない場合は，初期値不設定記号の **?** を付けてください。構造体名を使ってデータを定義する場合は，値の指定がないフィールドには構造体定義内のデフォルト値が割り当てられます。値の指定もなくデフォルト値もない場合は，**?** が使われます。

文字列をデフォルト値として指定する場合は注意を要します。文字列が長すぎて

指定したフィールド内におさまらない場合は切り捨てられます。逆に文字列の長さがフィールド幅に満たない場合は，MASMモードではフィールドにスペースが詰められ，Idealモードでは構造体定義の残りの文字列が使われます。したがって，構造体定義に適切な値を指定しておくことによって，文字列の設定を変えることができます。

構造体メンバの宣言の途中に，データ定義疑似命令のかわりに**STRUC**または**UNION**疑似命令を指定すると，構造体または共用体をネストすることができます。定義済みの構造体名を使うこともできます。

STRUCまたは**UNION**疑似命令によって構造体または共用体をネストした場合は，構造体メンバ演算子のピリオド(.)を使って，構造体を1レベルとしてアクセスすることができます。定義済み構造体名によって構造体をネストした場合は，複数のピリオド演算子を使えばマルチレベルの構造体にアクセスすることができます。

参照　**ENDS, UNION**

例

```
Ideal
model small
dataseg
struc B
    B1  DD  0
    B2  DB  ?
ends

struc A
    A1  DW  ?
    A2  DD  ?
    binst  B  <>
    struc
        D  DB  "XYZ"
        E  DQ  1.0
    ends
ends

ainst  A  <>
cinst  A  ?
dinst  A

codeseg
    mov  al,[ainst.binst.b2]
    mov  al,[ainst.d]
    mov  ax,[word cinst.binst.b1]
end
```

SUBSTR

機能　新しい文字列を既存文字列の副文字列として定義します。

モード　MASM51, Ideal

構文　*name* **SUBSTR** *string, position* [*,size*]

説明　*name*は，*position*で指定される位置から始まり，*size*で指定される長さの*string*内の文字列に付ける名前です。*string*内の先頭文字の*position*は1です。*size*を省略すると，*position*の位置から*string*内の最後の文字までを返します。

*string*には，次のどれかを指定できます。

■不等号カッコで囲まれた文字列引数（例 <*abc*>）
■定義済みテキストマクロ
■パーセント（%）で始まる式（評価されて数値文字列となる）

参照　**CATSTR, INSTR, SIZESTR**

例

```
N = 0AH
HEXC  SUBSTR <0123456789ABCDEF>,N + 1,1    ; HEXC = "A"
```

SUBTTL

機能　リストファイルにサブタイトルを設定します。

モード　MASM

構文　**SUBTTL** *text*

説明　サブタイトル*text*は，各ページの最上部，ソースファイル名の後，**TITLE**疑似命令で設定したタイトルの後に表示されます。
プログラム内には必要なだけいくつでも**SUBTTL**疑似命令を入れることができます。それぞれの疑似命令は次のページの最上部に表示されるサブタイトルを設定します。

参照　**%SUBTTL**

例

```
SUBTTL  ビデオドライバ
```

%SUBTTL

機能　リストファイルにサブタイトルを設定します。

モード　MASM, Ideal

構文　**%SUBTTL** *"text"*

説明　サブタイトル*text*は，各ページの最上部，ソースファイル名の後，**%TITLE**疑似命令で設定したタイトルの後に表示されます。サブタイトルの文字列は，2つの引用符（" "）で囲んでください。
プログラム内には必要なだけいくつでも**%SUBTTL**疑似命令を入れることができます。それぞれの疑似命令は次のページの最上部に表示されるサブタイトルを設定します。

参照　**SUBTTL**

例

```
%SUBTTL  "出力ルーチン"
```

%SYMS

機能　シンボルテーブルのリスト表示を有効にします。

モード　MASM, Ideal

構文　%SYMS

説明　ソースファイル内に**%SYMS**を指定すると，シンボルテーブルがリストファイルの最後に付けられます（シンボルテーブルは，ソースファイル内で定義されたすべてのシンボルの一覧を表示します)。

このモードは，ソースアセンブル開始時のデフォルトリストモードです。

参照　%NOSYMS

例

```
%SYMS          ; シンボルテーブルがリストファイル内に作られる
```

%TABSIZE

機能　リストファイルのタブ間隔を桁数で設定します。

モード　MASM, Ideal

構文　**%TABSIZE** *width*

説明　*width*は，リストファイルのタブ間隔（桁数）を指定します。タブ間隔のデフォルト値は8です。

参照　**%PAGE, %PCNT, %BIN, %TEXT**

例

```
%TABSIZE 4          ; 狭いタブ幅
```

%TEXT

機能　リストファイルのソースフィールドの幅を設定します。

モード　MASM, Ideal

構文　**%TEXT** *width*

説明　*width*は，リストファイルのソース行フィールド幅を桁数で指定します。ソース行の幅ががこの幅より大きい場合は，ソース行は切り捨てられます(**%TRUNC**指定時)。または次の行へ送られます（**%NOTRUNC**指定時)。

参照　**%BIN, %DEPTH, %NOTRUNC, %PCNT, %TRUNC**

例

```
%TEXT 80        ; ソースフィールド幅を80にセット
```

.TFCOND

機能　条件ブロックのリスティングモードを切り替えます。

モード　MASM

構文　**.TFCOND**

説明　特に指定のない限り，条件ブロックはリスト表示されません。**.TFCOND**によって条件ブロックのリスト表示が有効になります。**/X**コマンドラインオプションで条件ブロックのリスト表示が指定されている場合は，**.TFCOND**によってリスト表示は無効になります。ソースファイル内に**.TFCOND**が見つかるたびに，偽条件ブロックのリスト表示の有効/無効が交互に切り替わります。

参照　**%CONDS, .LFCOND, %NOCONDS, .SFCOND**

TITLE

機能　リストファイルにタイトルを設定します。

モード　MASM

構文　**TITLE** ***text***

説明　タイトル*text*は，各ページの最上部，ソースファイル名の後，**SUBTTL**疑似命令で設定したサブタイトルの前に表示されます。
TITLE疑似命令は，プログラム内に1回だけ使うことができます。

参照　**SUBTTL, %SUBTTL, %TITLE**

例

```
TITLE  ソートユーティリティ
```

%TITLE

機能　リストファイルにタイトルを設定します。

モード　MASM, Ideal

構文　**%TITLE** *"text"*

説明　タイトル*text*は，各ページの最上部，ソースファイル名の後，**%SUBTTL**疑似命令で設定したサブタイトルの前に表示されます。サブタイトルの文字列は，2つの引用符（" "）で囲んでください。

%TITLE疑似命令は，プログラム内に1回だけ使うことができます。

参照　**SUBTTL, %SUBTTL, TITLE**

例

```
%TITLE  "I/Oライブラリ"
```

%TRUNC

機能　リストファイル内の長すぎるフィールドを切り捨てます。

モード　MASM, Ideal

構文　**%TRUNC**

説明　**%TRUNC**は，**%NOTRUNC**の逆の働きをします。**%TRUNC**疑似命令は，オブジェクトコードフィールドとソース行フィールドを変更して，幅の広すぎるフィールドを切り捨てます。切り捨てられた情報は失われます。

参照　**%NOTRUNC**

例

```
%TRUNC
    DD 1,2,3,4,5        ; 一部のフィールドを表示
```

UDATASEG

機能　未初期化データセグメントの開始を定義します。

モード　MASM, Ideal

参照　**.CODE, .CONST, .DATA, .DATA?, .FARDATA, .FARDATA?, .MODEL, .STACK**

UFARDATA

機能　未初期化farデータセグメントの開始を定義します。

モード　MASM, Ideal

参照　**.CODE, .DATA, .FARDATA, .FARDATA?, .MODEL, .STACK**

```
name UNION
    fields
[name] ENDS
```

UNION

機能　共用体を定義します。

モード　MASM, Ideal

構文　MASMモード

name **UNION**
　fields
[*name*] **ENDS**

Idealモード

UNION [*name*]
　fields
ENDS [*name*]

説明　IdealモードとMASMモードにおける**STRUC**疑似命令の違いは，1行目の**UNION**と共用体名の順序，および2行目の**ENDS**とオプションの共用体名の順序が反対であることです。**UNION**疑似命令をネストしたり，**UNION**疑似命令と**STRUC**疑似命令を組み合わせて使うことができます。

UNIONは，そのすべてのメンバの，共用体の先頭からのオフセットが0である点を除いて**STRUC**と同様です。これによってオーバレイされる一組のフィールドを定義でき，共用体によって定義されたメモリ領域を別のシンボルおよびデータサイズで参照することが可能となります。共用体のサイズは，そのメンバのうちの最大のメンバのサイズになります。構造体（**STRUC**）の場合は，すべてのメンバのサイズの和が構造体のサイズとなります。

*name*は共用体の名前です。後でこのフォーマットの共用体としてメモリを確保する際に，この名前を使うことができます。

*fields*は共用体の構成フィールドを定義します。それぞれのフィールドのサイズは，通常のデータ定義疑似命令（**DB, DW**など）を使って定義します。フィールド名は，シンボル名と同様にプログラム内で固有でなくてはなりません。

フィールド定義において，データ定義疑似命令の後にデフォルト値を指定することができます。これは各データを初期化することと同じです。デフォルト値を指定しない場合は，未初期化記号の **?** を付けてください。共用体名を使ってデータ

を定義する場合は，値の指定がないフィールドには共用体定義内のデフォルト値が割り当てられます。値の指定もなくデフォルト値もない場合は，?が使われます。

文字列をデフォルト値として指定する場合は注意を要します。文字列が長すぎて指定したフィールド内に納らない場合は切り捨てられます。逆に文字列の長さがフィールド幅に満たない場合は，フィールドにスペースが詰められます。

共用体メンバの宣言の途中にデータ定義疑似命令のかわりに**STRUC**または**UNION**疑似命令を指定すると，構造体または共用体をネストすることができます。**STRUC**または**UNION**疑似命令によって構造体または共用体をネストした場合は，構造体メンバ演算子のピリオド(.)を使って，構造体を1レベルとしてアクセスすることができます。定義済み共用体名によって共用体をネストした場合は，複数のピリオド演算子を使えばマルチレベルの構造体と共用体にアクセスすることができます。

参照　**ENDS**

例

```
UNION B
    bmem1   DW   ?
    bmem2   DB   ?
ENDS
UNION A
    B       DW   ?                    ; 第1フィールド...オフセット0
    C       DD   ?                    ; 第2フィールド...オフセット0
bunion  B   <>                        ; 初期値0
  STRUC
    D      DB   "XYZ"                 ; オフセット0
    E      DQ   1,0                   ; オフセット1
  ENDS
ENDS
ainst    A  <>                        ; A型の共用体を確保
    mov  al,[ainst.bunion.bmem1]      ; マルチレベル
    mov  al,[ainst.d]                 ; 1レベル
```

USES

機能　プロシージャのレジスタの使い方を示します。

モード　MASM, Ideal

構文　**USES** *register* [*,register*] ...

説明　**USES**は，**PROC/ENDP**のペアで挟まれたブロック内で使うことができ，プロシージャの入口でプッシュしたいレジスタと，プロシージャの出口でポップしたいレジスタを指定します。
*register*には次のどれかを指定できます。

AX	CX	DS	ES
BX	DI	DX	SI

80386プロセッサを有効にしている場合は，次の32ビットレジスタを指定することもできます。

EAX	ECX	EDI	EBX
EDX	ESI		

PROC疑似命令での複数レジスタの指定では，区切り文字としてスペースを使いますが，**USES**疑似命令では複数のレジスタ名はカンマで区切ります。
上記レジスタは，**PROC**疑似命令でも指定できます。この疑似命令はプロシージャ宣言を見やすくするので，プロシージャの入口と出口でのセットアップを定義するマクロの内部で指定することができます。
この疑似命令は，プロシージャ内のコードを生成する最初の命令より先に指定してください。

参照　**ARG, LOCALS, PROC**

例

```
COPY10 PROC
USES  cx,si,di
      mov  cx,10
REP   movsb
RET                   ; SI,DIレジスタをポップする
MYPROC ENDP
```

WARN

機能　警告メッセージを有効にします。

モード　MASM, Ideal

構文　**WARN** *warnclass*

説明　**WARN**に*warnclass*を指定しない場合は，すべての警告メッセージが有効になります。**WARN**の後に警告メッセージ識別シンボルを指定すると，そのメッセージのみ有効になります。次に示すように，各メッセージには3文字の識別シンボルが割り当てられています。

ASS	Assumes segment is 16-bit (セグメントは16ビットであると想定します)
BRK	Bracket needed (大カッコが必要です)
ICG	Inefficient code generation (コードを生成することができません)
LCO	Location counter overflow (ロケーションカウンタオーバフロー)
OPI	Open IF conditional (IF条件文が閉じていません)
OPP	Open procedure (プロシージャが閉じていません)
OPS	Open segment (セグメントが閉じていません)
OVF	Arithmetic overflow (算術オーバフロー)
PDC	Pass-dependent construction (パスに依存する制御構造です)
PRO	Write-to-memory in protected mode needs CS override (保護モードでのメモリ書き込みにはCSオーバライドが必要です)

RES　Reserved word warning
（予備の警告メッセージです）

TPI　Turbo Pascal illegal warning
（Turbo Pascalの不正な警告メッセージです）

PQK　Assuming constant for const warning
（[const] 警告メッセージに定数を想定します）

ALN　Segment alignment
（セグメントのアラインメント）

これらのシンボルは，/**W**コマンドラインオプションで使うものと同じです。

参照　**NOWARN**

例

```
WARN OVF               ; 算術オーバフローの警告メッセージを有効にする
DW  1000h * 1234h      ; 警告メッセージが表示される
```

.XALL

機能　コードまたはデータを生成するマクロ展開のみリストします。

モード　MASM

参照　**.LALL, %MACS, %NOMACS, .SALL**

.XCREF

機能　クロスリファレンスリスト（CREF）の表示を無効にします。

モード　MASM

参照　**%CREF, .CREF, %NOCREF**

.XLIST

機能　リスト表示を無効にします。

モード　MASM

参照　**%LIST, .LIST, %NOLIST**

付録A

Turbo Assemblerの構文のまとめ

この付録では，バッカス・ナウア記法(BNF)に基づいて，Turbo AssemblerのMASMモードとIdealモードの式構文をまとめます。

注意：以降のセクションでは，省略形(...)は要素を必要なだけ何回でも繰り返して指定できることを表わします。

語意に関する文法

valid_line

■*white_space valid_line*

■*puctuation valid_line*

■*number_string valid_line*

■*id_string valid_line*

■*null*

white_space

■*space_char white_space*

■*space_char*

space_char

■すべての制御文字，128より大きい文字，

id_string

■*id_char id_strng2*

id_strng2

■*id_chr2 id_strng2*

■*null*

id_char

■$, %, _, ?, 英文字

id_chr2

■*id_chars plus numerics*

number_string

■*num_string*

■*str_string*

num_string

■*digits alphanums*

■*digits'.'digits exp*

■*digits exp*　　; *DD, DQ, DTのMASMモードのみ*

digits

■*digit digits*

■*digit*

digit

■0～9

alphanums

■*digit alphanum*

■*alpha alphanum*

■*null*

alpha

■英文字

exp

■*E + digits*

■*E - digits*

■*E digits*

■*null*

str_string

■引用符で囲まれた文字列。引用符自身は2つの引用符で表わす。

punctuation

■space_char, id_char, ' " ', " ' ", 数字以外のすべて

MASMモードとIdealモードでは，ピリオド(.)の扱いが異なります。この文字は，MASMモードの浮動小数点数には必要ありません。Idealモードではシンボル名の一部として使うことはできません。MASMモードでは，シンボル名の先頭文字や構造体メンバの選択子としての句読文字に使われます。

次にピリオド（.）の使用規則を示します。

1. Idealモードでは，常に句読文字として使われます。
2. MASMモードでは，以下の場合にIDの先頭文字として扱われます。
   a. 行の先頭文字に使われた場合または**EXTRN, PUBLIC**等のシンボル内で使われた場合は，後続文字が*id_chr2*であれば後続シンボルと結合する（上記規則に従う)。
   b. 行の先頭文字以外で使われた場合またはピリオドを含むシンボルが定義済みシンボルと重なる場合は，ピリオドは後続シンボルの先頭と結合する。

MASMモードの式に関する文法

mexpr1

■*'SHORT' mexpr1*

■*'.TYPE' mexpr1*

■*'SMALL' mexpr1*（*16*ビットオフセットキャスト［*386*のみ］)

■*'LARGE' mexpr1*（*32*ビットオフセットキャスト［*386*のみ］)

■*mexpr2*

mexpr2

■*mexpr3 'OR' mexpr3 ...*

■*mexpr3 'XOR' mexpr3 ...*

■*mexpr3*

mexpr3

■*mexpr4 'AND' mexpr4 ...*

■*mexpr4*

mexpr4

■*'NOT' mexpr4*

■*mexpr5*

mexpr5

■*mexpr6 'EQ' mexpr6 ...*

■*mexpr6 'NE' mexpr6 ...*

■*mexpr6 'LT' mexpr6 ...*

■*mexpr6 'LE' mexpr6 ...*

■*mexpr6 'GT' mexpr6 ...*

■*mexpr6 'GE' mexpr6 ...*

■*mexpr6*

mexpr6

■*mexpr7 '＋' mexpr7 ...*

■*mexpr7 '-' mexpr7 ...*

■*mexpr7*

mexpr7

■*mexpr8 '＊' mexpr8 ...*

■*mexpr8 '｜' mexpr8 ...*

■*mexpr8 'MOD' mexpr8 ...*

■*mexpr8 'SHR' mexpr8 ...*

■*mexpr8 'SHL' mexpr8 ...*

■*mexpr8*

mexpr8

- ■*mexpr9 'PTR' mexpr8*
- ■*mexpr9*
- ■*'OFFSET' mexpr8*
- ■*'SEG' mexpr8*
- ■*'TYPE' mexpr8*
- ■*'THIS' mexpr8*

mexpr9

- ■*mexpr10 ':' mexpr10 ...*

mexpr10

- ■*'+' mexpr10*
- ■*'-' mexpr10*
- ■*mexpr11*

mexpr11

- ■*'HIGH' mexpr11*
- ■*'LOW' mexpr11*
- ■*mexpr12*

mexpr12

- ■*mexpr13 mexpr13 ...* ('[' または'('があるときのみ暗黙に追加される)
- ■*mexpr12 mexpr13 '.' mexpr8*

mexpr13

- ■*'LENGTH' id*
- ■*'SIZE' id*
- ■*'WIDTH' id*
- ■*'MASK' id*
- ■'('*mexpr1* ')'
- ■'['*mexpr1*']'
- ■*id*
- ■*const*

Idealモードの式に関する文法

pointer

- *'SMALL' pointer1*（16ビットオフセットキャスト［386のみ］）
- *'LARGE' pointer1*（32ビットオフセットキャスト［386のみ］）
- *type 'PTR' pointer*
- *type 'LOW' pointer*（下位階級の操作）
- *type 'HIGH' pointer*（上位階級の操作）
- *type pointer*
- *pointer2*

type

- *'UNKNOWN'*
- *'BYTE'*
- *'WORD'*
- *'DWORD'*
- *'QWORD'*
- *'PWORD'*
- *'TBYTE'*
- *'SHORT'*
- *'NEAR'*
- *'FAR'*
- *struct_id*
- *'TYPE' pointer*

pointer2

- *pointer3 '.' id*（構造体メンバの操作）
- *pointer3*

pointer3

- *expr ':' pointer3*
- *expr*

expr

■*'SYMTYPE' expr*（シンボル型の操作）

■*expr2*

expr2

■*expr3 'OR' expr3 ...*

■*expr3 'XOR' expr3 ...*

■*expr3*

expr3

■*expr4 'AND' expr4 ...*

■*expr4*

expr4

■*'NOT' expr4*

■*expr5*

expr5

■*expr6 'EQ' expr6 ...*

■*expr6 'NE' expr6 ...*

■*expr6 'LT' expr6 ...*

■*expr6 'LE' expr6 ...*

■*expr6 'GT' expr6 ...*

■*expr6 'GE' expr6 ...*

■*expr6*

expr6

■*expr7 '+' expr7 ...*

■*expr7 '-' expr7 ...*

■*expr7*

expr7

■*expr8 ' * ' expr8 ...*

■*expr8 ' | ' expr8 ...*

■*expr8 'MOD' expr8 ...*

■*expr8 'SHR' expr8 ...*

■*expr8 'SHL' expr8 ...*

■*expr8*

expr8

■'+'*expr8*

■'–' *expr8*

■*expr9*

expr9

■*'HIGH' expr9*

■*'LOW' expr9*

■*expr10*

expr10

■*'OFFSET' pointer*

■*'SEG' pointer*

■*'SIZE' id*

■*'LENGTH' id*

■*'WIDTH' id*

■*'MASK' id*

■*id*

■*const*

■'('*pointer* ')'

■'[*pointer*']' (常に'*contents-of* 'を意号する)

付録B

互換性の問題

Turbo AssemblerのMASMモードは，MASMのバージョン4.0と互換性があります。さらにMASMのバージョン5.0と5.1で追加されたすべての拡張機能をサポートしています。ただしアセンブリ言語には公式仕様がなく，MASMはバージョン間で互換性がないため，100%の互換性は理想にすぎません。

ほとんどのプログラムでは，MASMのバージョン4.0または5.1のかわりとして直接Turbo Assemblerを使っても何も問題はありません。Turbo Assemblerは，MASMにないメッセージを出力することがときどきありますが，これはMASMで検出しないエラー文をTurbo Assemblerが検出するためです。次に例を示します。

```
ABC  EQU [BP+2]
     PUBLIC  ABC
```

MASMはこのコードをエラーとせず意味のないオブジェクトファイルを生成します。しかしTurbo Assemblerは，このような意味不明のコードをエラーとして認識します。

Turbo Assemblerのアセンブル処理でトラブルが発生したら，**QUIRKS**疑似命令を使ってください。たとえば，コマンドライン

```
TASM /JQUIRKS MYFILE
```

でプログラムが正しくアセンブルできたら，ソースファイルの先頭に**QUIRKS**疑似命令を入れてください。この付録を復習して，ソースファイルのどのコードが**QUIRKS**疑似命令を必要とするかを決めてください。そして**QUIRKS**を使わないで済むように，そのコード行を書き直してください。

MASMのバージョン5.1の特定の機能を使っている場合は，ソースファイル中に**MASM51**疑似命令を入れる必要があります。この疑似命令の機能については，この付録で後述します。

環境変数

ボーランド社の他の言語ソフトで使われている手法に準じて，Turbo Assemblerは環境変数を使ってデフォルトのオプションをコントロールすることはありません。そのかわり，コンフィギュレーションファイル内にデフォルトオプションを置いて，プロジェクトごとに異なったコンフィギュレーションファイルをセットアップすることができます。この手順はユーザーズガイドの第3章を参照してください。

すでに**INCLUDE**または**MASM**環境変数を使ってMASMの動作を設定してある場合は，Turbo Assembler用のコンフィギュレーションファイルを作らなければなりません。**MASM**変数を使って指定したオプションは，コンフィギュレーションファイル内にそのまま入れることができます。**INCLUDE**変数を使って指定したディレクトリは，コマンドラインオプション/**I**を使ってコンフィギュレーションファイル内に入れる必要があります。

マイクロソフトのバイナリ浮動小数点フォーマット

MASMの従来のバージョンは，デフォルトで浮動小数点数を生成しますが，そのフォーマットはIEEEの標準浮動小数点フォーマットと互換性がありません。MASMのバージョン5.1はデフォルトでIEEEの浮動小数点数を生成し，従来フォーマットを指定するための**.MAFLOAT**疑似命令を備えています。

Turbo Assemblerは従来の浮動小数点フォーマットをサポートしないので，**.MSFLOAT**疑似命令はありません。

Turbo AssemblerのQuirksモード

MASMの機能の中には，本質的に問題のある機能があります。これらの問題機能はTurbo AssemblerのMASMモードにはありません。しかしプログラマはときにそのような〝癖″のある機能を利用したい場合もあります。Turbo Assemblerは，このようなMASMの問題機能をエミュレートするためのQuirksというモードを備えています。

Quirksモードにするためには，ソースファイルに**QUIRKS**キーワードを入れておくか，Turbo Assembler起動時にコマンドラインオプションの/**JQUIRKS**を使います。

以下のコードは，Turbo AssemblerのMASMモードではエラーとなります(エラーメッセージを出す）が，Quirksモードではエラーとなりません。

セグメントレジスタ間でのバイト転送

MASMはセグメントレジスタとメモリとの間のデータ転送の際にオペランドサイズをチェックしません。たとえば，MASMは次のコードを正常に受け付けます。

```
SEGVAL  D...  ?
USEFUL  DB    ?
   mov  SEGVAL,es          ; "USEFUL"の部分を上書きする
```

これは本来明らかにプログラミングエラーですが，*USEFUL*が不正でもプログラムの動作に影響しないため正常に動作します。Quirksモードを使うよりむしろ*SEGVAL*を**DW**に定義し直すべきです。

farラベルまたはfarプロシージャへのエラーnearジャンプ

MASMでは，ジャンプ先が**FAR PTR**でオーバライドされているか否かにかかわらず，同一セグメント内のアドレスへのfarジャンプに対してnearまたはshortジャンプを生成します。

```
CODE SEGMENT
     JMP ABC
     JMP FAR PTR ABC      ; farジャンプを生成しない
ABC  LABEL FAR
CODE ENDS
```

Turbo Assemblerでは，ジャンプ先がfarポインタであることを指定すれば，通常farJMP命令コードを生成します。MASMと同じように動作させ，同一セグメント内アドレスへのジャンプ命令を常にshortまたはnearジャンプとして扱う場合は，Quirksモードにする必要があります。

=疑似命令とEQU疑似命令の型情報の消失

次のコード例を見てください。

```
X DW 0
Y = OFFSET X
    mov ax,Y
```

このコードからMASMは**MOV　AX,** [X] を生成し，Turbo AssemblerはMOV **AX, OFFSET X**を生成します。このようなことが起こる理由は，MASMでは=疑似命令の右辺の式に関するすべての情報を保存しないためです。

EQU疑似命令を使って数式用シンボルを定義するときも同様なことが起こります。

セグメントアラインメントのチェック

MASMで**ALIGN**疑似命令を使えば，セグメントアラインメントよりも厳密なアラインメントを指定することができます。次に例を示します。

```
CODE SEGMENT WORD
     ALIGN   4      ; セグメントはワードでアラインメントされている
CODE ENDS
```

このような**ALIGN**の使い方は危険なので避けてください。この例では，リンカが**CODE**セグメントのこの部分を他のモジュール内の同一名セグメントと結合すると，**ALIGN**疑似命令の効力は無効となります。またモジュール内のセグメント部分がワードアラインメントよりも適切にアラインメントされる保証はどこにもありません。

このような基本的に意味のないコードを受け付けるためには，Quirksモードにする必要があります。

符号付き即値の算術演算命令と論理演算命令

MASMのバージョン4.0は，算術演算命令の即値オペランドは符号付きで拡張します。Turbo AssemblerがQuirksモードになっていない場合は，論理演算命令に対しても符号付き拡張が行なわれます。その結果コンパクトで高速なコードが生成できますが，その命令があるコードセグメントのサイズは変化します。これは自分を変更するプログラムや生成された命令の長さをおおよそ知っているコードに関しては，問題が生じます。次に示すコードでは，1行目についてはMASMとTurbo Assemblerの両者が正常にコードを生成しますが，2行目についてはTurbo Assemblerは正常に生成するけれども，MASMのバージョン4.0は正しい生成を行ないません。

```
ADD   ax,-1        ; MASMでもTurbo Assemblerでも符号付き拡張する
XOR   cx,0FFFFh    ; MASM 4.0はワード即値を使う．Turbo Assemblerは，
                   ; Quirksモード以外ではバイトの符号付き拡張を行なう．
```

この例の**XOR**命令に対しては，MASMのバージョン4.0は81 F1 FFFFのバイトシーケンスを生成し，Turbo Assemblerはより短かい互換コード83 F1 FFを生成します。

Turbo Assemblerを使ってXOR CX, 0FFFFH命令の同じバイトシーケンスを生成させるには，Quirksモードを使わなくてはなりません。

MASMのバージョン5.1の機能

MASMのバージョン5.1で導入されている新しい機能のいくつかは，Turbo Assemblerにおいても利用することができます。**MASM51**疑似命令で使用可能になる機能もあります。MASMのバージョン5.1の機能とTurbo Assemblerの機能（Turbo Assemblerではパワーアップされています）は，少し前の「Turbo AssemblerのQuirksモード」の節で説明しました。

Turbo Assemblerは，MASMモードでソースファイルのアセンブルを開始します(このモードではMASM 5.1の機能は使用できません)。プログラムに**MASM**疑似命令と**NOMASM51**疑似命令を入れた場合も同じです。

以下に示す拡張機能のリストは，このマニュアルの第2章の「演算子」および第3章の「疑似命令」で，より詳しく説明されています。

MASMのバージョン5.1の以下の機能は，常時利用することができます。

■**PROC**疑似命令に対するパラメータとローカル引数
■**.TYPE**演算子の拡張機能
■**COMM**疑似命令の拡張機能
■**.CODE**は**CS ASSUME** でカレントセグメントを設定
■**.MODEL**疑似命令の高級言語サポート機能
■全コマンドラインオプションのリスト機能（**/LA** ）
■**DW, DD, DF**疑似命令のデバッグ情報の追加機能
■疑似命令の**ELSEIF**ファミリ
■**@Cpu**疑似命令と**@WordSize**シンボル
■テキストマクロ用の%式演算子
■**DW, DD, DF**のデバッグ情報拡張機能

MASM51を使うと，次の機能を利用することができます。

■**SUBSTR, CATSTR, SIZESTR, INSTR** 疑似命令
■円記号¥（またはバックスラッシュ＼）による行継続
■**LOCAL**疑似命令の代替引数構文

MASM51と**QUIRKS**の両方を使うと，次の機能を利用することができます。

■@@で定義され，**@F**と**@B**で参照されるローカルラベル
■**PROC**内における変数の再定義
■C言語用の**PROC**は，先頭に下線記号を付けることですべて**PUBLIC**となる

MASMバージョン5.1またはQuirksモードの機能

MASMバージョン5.1の機能には，逆にTurbo Assemblerのある種の機能に影響を与えるものがあります。この付録ではTurbo AssemblerにおけるMASMバージョン5.1の代替機能を説明しました。これらの機能を使うには，**QUIRKS**疑似命令でQuirksモードを指定して，なおかつ**MASM51**疑似命令でMASMのバージョン5.1の機能を有効にしなければなりません。

付録C

Turbo Assemblerの重要項目

Turbo Assemblerは，MASMとの互換性の他にも，一般のMASM形式の命令で利用できる多数の拡張機能を備えています。これらの拡張機能は，IdealモードでもMASMモードでも使うことができます（Idealモードの詳細は，**ユーザーズガイド**の第12章を参照してください）。

この付録ではそれぞれの拡張機能を説明し，各項目の参照先を示します。

拡張されたコマンドラインの構文

Turbo Assemblerのコマンドラインの構文（MASMのそれのスーパセットになっています）は，大幅に改善されています。ファイルを1つ1つ指定するか，ワイルドカード（*と?）を指定することによって，複数ファイルのアセンブルが可能となっています。またファイルをグループ化することによって，コマンドラインオプションの複数セットとファイルの複数セットを一対一対応させることができます。Turbo Assemblerのコマンドラインオプションの詳しい解説は，**ユーザーズガイド**の第3章にあります。

GLOBAL疑似命令

GLOBAL疑似命令を使うと，変数に**EXTRN**と**PUBLIC**の両方を指定したことになります。つまり，**GLOBAL**定義をヘッダファイル内で行ない，このヘッダファイルをすべてのソースファイルにインクルードしておけば，ただ1つのモジュール内でデータを定義することができます。**GLOBAL**疑似命令で定義されたデータは初期化することができるため，大幅に柔軟性が向上します（**COMM**疑似命令で定義されたデータは初期化することができません）。

ユーザーズガイド第5章の「GLOBAL疑似命令」の節にこの疑似命令の使い方の説明があります。**GLOBAL**を使ったデータ定義方法については，このマニュアルの第3章を参照してください。

ローカルシンボル

LOCALSと**NOLOCALS**の疑似命令を使えば，2つの単価記号（@@）で始まるシンボルを特定のコードブロックに対してローカルにすることができます。

ローカルシンボルの詳細は，**ユーザーズガイド**第10章の「ローカルラベル」の節を参照してください。このマニュアルの第3章でもこれら2つの疑似命令について説明しています。

条件ジャンプの拡張機能

JUMPSと**NOJUMPS**疑似命令は，条件ジャンプを拡張して条件反転を行なうか否かとnearジャンプ命令をコントロールします。条件反転によって，-128～＋127バイトの範囲を越えるアドレスへの条件ジャンプが可能となります。

これらの疑似命令の使い方については，**ユーザーズガイド**第10章の「自動ジャンプサイズ決定」の節を参照してください。

Idealモード

Turbo AssemblerのIdealモードでは，式や命令オペランドをより合理的に扱うことが可能となります。わずかの規則を憶えるだけで，複雑な命令オペランドを簡単に扱うことができます（Idealモードの役割については，**ユーザーズガイド**の第12章を参照してください）。

UNION疑似命令/STRUCのネスティング

共用体は，**STRUC**疑似命令で定義される構造体と似ています。ただし，全メンバのデータ構造の先頭からのオフセットは0であり，すべてのメンバは効率よくオーバレイされている点が構造体と異なります。

Turbo Assemblerでは，**STRUC**をネストしたり，**UNION**と結合したりすることができます。**ユーザーズガイド**第10章の「STRUC疑似命令」の節にこの疑似命令の使い方の説明があります。また，本マニュアルの第3章に**STRUC**と**UNION**疑似命令の詳細説明があります。

ARGとLOCAL疑似命令

ARGと**LOCAL**疑似命令は，特定関数にローカルでBPレジスタに依存しない変数名を定義します。これらの変数名を使えば，高級言語から渡された引数の参照，ルーチンの呼び出し，プロシージャ用のスタック一とき領域の確保などを簡単に行なうことができます。

ユーザーズガイド第6章の「パラメータの引き渡し」の節にこの疑似命令の使い方の説明があります。また，このマニュアルの第3章に**ARG**と**LOCAL**疑似命令の詳細説明があります。

EMULとNOEMUL疑似命令

EMULと**NOEMUL**疑似命令を使って，浮動小数点命令がエミュレート命令かコプロセッサの実命令かを選択することができます。1つのソースファイル内で，必要なだけ何回でもエミュレート命令と実浮動小数点命令を切り替えることができます。

明示的なセグメントオーバライド

Turbo Assemblerでは，オーバライド疑似命令の**SEGCS,SEGDS,SEGES,SEGSS,SEGFS,SEGGS**のどれかを使って，特定の命令に対して明示的にセグメントオーバライドを指定することができます。これらの疑似命令の機能は，**REP**や**LOCK**オーバライドと非常によく似ています。

ユーザーズガイド第10章の「セグメントオーバライドプレフィクス」の節にこの疑似命令の使い方の説明があります。

定数セグメント

Turbo Assemblerでは，セグメントの値を与える必要があるときはいつでも定数を使うことができます。セグメントに定数を追加することもできます。次に例を示します。

```
      JMP     FAR PTR 0FFFFh:0       ; ROM BIOSへジャンプ
LOWDATA SEGMENT AT 0
      ASSUME DS:LOWDATA+40h          ; DSはBIOSデータ領域をポイント
      mov     ax,DS:[3FH]            ; BIOSデータ領域からワードを読み込む
LOWDATA ENDS
```

ユーザーズガイド第10章の「SEGMENT疑似命令」の節に詳細な説明があります。

386モードにおける拡張LOOP命令

Turbo Assemblerで80386用のコードを書く際は，CXまたはECXのどちらを**LOOP**命令のカウンタとして使うかを明示的に指定することができます。

ユーザーズガイド第11章の「LOOPとJCXZの32ビットバージョン」の節にこの命令の使い方の説明があります。

拡張リスト制御機能

Turbo Assemblerのファイルのリスト機能では，リストの書式と内容の指定機能が一段と充実しています。**インクルード**ファイルをリストするか否か，リスト制御状態のプッシュとポップ（退避と復帰），リストのフィールド幅の指定（切り捨てるか次行に送るかの指定を含む）などをコントロールすることができます。

ユーザーズガイドの第5章に使用できるすべてのオプションの詳細な説明があります。

代替疑似命令

Turbo Assemblerは，多くの疑似命令（特にピリオド(.)で始まる疑似命令）に対する代替キーワードを備えています。リスト制御疑似命令の代替疑似命令は，パーセント記号(%)で始まります。プロセッサ指定疑似命令の代替疑似命令は，*P*で始まります。

Turbo Assemblerがサポートするすべての疑似命令の一覧は，このマニュアルの第3章を参照してください。

定義済み変数

Turbo Assemblerは，ソースファイルから値をアクセスできる多くの定義済み変数を持っています。これらの変数には，MASMのバージョン5.0との互換性を保つためにサポートされている定義済み変数に加えて，**??Date**，**??Time**，**??Filename**，**??Version**が含まれます。

これらの変数の定義については，このマニュアルの第1章「定義済みシンボル」を参照してください。

MASMのバージョン5.0と5.1における改良点

Turbo Assemblerは，MASM 4.0からMASM 5.0と5.1へのバージョンアップの際に拡張されたすべての機能をサポートします。これらの拡張機能に慣れていない場合は，それらの説明を読んでください（以下に重要項目とその参照先の一覧を示します）。

■**80386のサポート——ユーザーズガイド**の第11章の「80386」
■**簡略セグメント疑似命令——ユーザーズガイド**の第11章の「簡略セグメント疑似命令と80386のセグメント型」
■**文字列と同等なシンボル——ユーザーズガイド**の第5章の「イコール定義」
■**RETFとRETN——ユーザーズガイド**の第4章の「サブルーチンの動作」
■**共用変数**——本マニュアルの第3章の**COMM**疑似命令
■**明示的インクルードライブラリファイル**——本マニュアルの第3章の**INCLUDELIB**疑似命令
■**より柔軟なデータ構造の定義——ユーザーズガイド**の第11章の「構造体と共用体」
■**定義済み変数——ユーザーズガイド**の第4章の「簡略セグメント疑似命令」と第10章

改善されたSHLとSHRの扱い

SHLと**SHR**を算術式の一部として使う場合は，MASMでは負のシフトカウントは指定できません。しかしTurbo Assemblerでは，負のシフトカウントを指定でき，反対の型のシフト処理を実行することができます。たとえば16 **SHL**-2と16 **SHR** 2は同じです。

付録D

Turbo Assemblerのユーティリティ

Turbo Assemblerには，5つの強力なスタンドアロンユーティリティが用意されています。これらのスタンドアロンユーティリティは，Turbo Assemblerのファイルだけでなく，他のモジュールにたいしても使用することができます。

■MAKE（スタンドアロンプログラムマネージャ，TOUCHを含む）
■TLINK（Turbo Linker）
■TLIB（Turbo Libralian）
■GREP（ファイル検索ユーティリティ）
■OBJXREF（オブジェクトモジュールクロスリファレンスユーティリティ）
■TCREF（クロスリファレンスユーティリティ）

この付録では，例題を用いて，各ユーティリティの機能と使い方を説明します。

スタンドアロンMAKEユーティリティ

Turbo Assemblerの強力な威力と高度の柔軟性とにより，多数のソースファイルやオブジェクトファイルから構成される大きくて複雑なプログラムを扱うことができます。その反面，この大きな柔軟性のために，どのファイルからどのファイルが作成されるか，またあるファイルを作成するためにどのファイルが必要か憶えていなければなりません。なぜなら，どれかのソースファイルを変更すると，それにより影響を受けるすべてのファイルを再アセンブルして，全体を再リンクしなければならないからです。もちろん，どれかのファイルを変更するたびに，すべてのファイルを再アセンブルして，再リンクすることもできます。しかしこの方法では，プログラムのサイズが大きくなるにつれて，ますます時間がかかるようになります。それではどうすればよいでしょうか。

答えは簡単です。MAKEを使うことです。Turbo AssemblerのMAKEは，インテリジェントなプログラム管理ユーティリティです。適正な指示を与えると，プログラムを最新バージョンに更新するために必要なすべての作業を行ないます。実際には，MAKEはそれ以上のことを行なうことができます。たとえば，バックアップを作成したり，複数のサブディレクトリからファイルを読み込んだり，プログラムが使用するデータファイルが変更されたときにそのプログラムを自動的に実行したりすることができます。MAKEを使うにつれて，プログラム開発の管理のための新しい利用の方法が見つかるでしょう。

この章では，スタンドアロンのMAKEを，Turbo AssemblerおよびTLINKとともに使用するときの使い方について説明します。

簡単な例

まず，MAKEがプログラムの更新にどんなに便利か，例を用いて説明します。いま近くの星の情報を表示するためのプログラムを書いているとしましょう。そのプログラムをGETSTARSとします。このプログラムは星のリストが入っているテキストファイルを読み込み，それらのデータを使って処理を行ない，バイナリデータファイルを作成して結果をその中に入れるものとします。

GETSTARSは，STARDEFS.INCファイルの中のいくつかの定義と，STARLIB.ASMの中のどれかのルーチンを使うとします。またプログラムGETSTARSは，次の3つのファイルに分割されているものとします。

■GSPARSE.ASM
■GSCOMP.ASM
■GETSTARS.ASM

最初の2つのファイルGSPARSEとGSCOMPは，それぞれ対応するインクルードファイル（GSPARSE.INCとGSCOMP.INC）を持つとします。また3番目のファイルGETSTARS.ASMには，プログラムの本体が入っているとします。さらに3つのファイルのうちGSCOMP.ASMとGETSTARS.ASMだけが，STARLIBのルーチンを使うものとします。

各アセンブラファイルが必要とするインクルードファイルは次のとおりです。

.ASMファイル	インクルードファイル
STARLIB.ASM	なし
GSPARSE.ASM	STARDEFS.INC
GSCOMP.ASM	STARDEFS.INC, STARLIB.INC,
GETSTARS.ASM	STARDEFS.INC, STARLIB.INC, GSPARSE.INC GSCOMP.INC

GETSTARS.EXE（ミディアムモデルとする）を作成するには，次のコマンドラインを使うことができます。

```
tasm /t /ml /s starlib
tasm /t /ml /s gsparse
tasm /t /ml /s gscomp
tasm /t /ml /s getstars
tlink starlib gsparse gscomp getstars, getstars, getstars, lib¥math lib¥io
```

上の説明から，ファイルの間には次のような依存関係があることがわかります。

■GSPARSE, GSCOMP, GETSTARSは，すべてSTARDEFS.INCに依存します。言い換えれば，STARDEFS.INCに変更を加えると，この3つのファイルはすべて再アセンブルしなければなりません。
■STARLIB.INCに変更を加えると，GSCOMPとGETSTARSを再コンパイルしなければなりません。
■GSPARSE.INCに変更を加えると，GETSTARSを再アセンブルしなければなりません。またGSCOMP.INCに変更を加えても，GETSTARSを再アセンブルしなければなりません。

■もちろんソースコードファイル(STARLIB.ASM, GSPARSE.ASMなど)に変更を加えると，そのファイルを再アセンブルしなければなりません。

■どれかのファイルを再アセンブルしたら，最後にリンクをやり直さなければなりません。

この依存関係を憶えておくのは困難です。もしSTARLIB.INCに変更を加えて, GETSTARS.ASMを再アセンブルし, GSCOMP.ASMを再アセンブルするのを忘れたらどうなるでしょうか。もちらんプログラムは正しく動作しないでしょう。どれかのファイルを変更したとき再アセンブルが必要となるファイルを再アセンブルするのを忘れないように, 前記の4回のアセンブルと1回のリンクを.BATファイルを使って行なうこともできます。しかしこの方法では，いつでもすべてのファイルを再アセンブルして，再リンクすることになります。MAKEを使うことによって，この問題を簡単に解決することができます。次にその方法を説明します。

メイクファイルの作成

メイクファイルは，ファイルの依存関係を示すリストと，その依存関係にしたがってアセンブルまたはリンクを行なうコマンドを組み合わせたものです。

例として，前の節で説明したファイルの依存関係にしたがってメイクファイルを作成すると，次のようになります。

```
getstars.exe: getstars.obj gscomp.obj gsparse.obj starlb.obj
        tlink starlib gsparse gscomp getstars, getstars,     ¥
              getstars, lib¥math lib¥io

getstars.obj: getstars.asm stardefs.inc starlib.inc gscomp.inc gsparse.inc
        tasm /t /ml /s getstars.asm

gscomp.obj: gscomp.asm stardefs.inc starlib.inc
        tasm /t /ml /s gscomp.asm

gsparse.obj: gsparse.asm stardefs.inc
        tasm /t /ml /s gsparse.asm

starlib.obj: starlib.asm
        tasm /t /ml /s starlib.asm
```

これは前の節で説明したのと同じことを記述しているだけです。ただし順番が逆になっていることに注意してください。MAKEは，このファイルを次のように解釈します。

■ファイルGETSTARS.EXEは，4つのファイルGETSTARS.OBJ, GSCOMP.OBJ, GSPARSE.OBJ, STARLIB.OBJに依存する。この4つのファイルのどれかが変更され

たら，GETSTARS.EXEは，指定されたTLINKコマンドを使って再リンクしなければならない。

■ファイルGETSTARS.OBJは，5つのファイルGETSTARS.ASM，STARDEF.INC，STARLIB.INC，GSCOMP.INC，GSPARSE.INCに依存する。この5つのファイルのどれかが変更されたら，GETSTARS.OBJは，指定されたTASMコマンドを使って再アセンブルしなければならない。

■ファイルGSCOMP.OBJは，3つのファイルGSCOMP.ASM，STARDEFS.INC，STARLIB.INCに依存する。この3つのファイルのどれかが変更されたら，GSCOMP.OBJは，指定されたTASMコマンドを使って再アセンブルしなければならない。

■ファイルGSPARSE.OBJは，2つのファイルGSPARSE.ASM，STARDEFS.INCに依存する。この2つのファイルのどれかが変更されたら，GSPARSE.OBJは，指定されたTASMコマンドを使って再アセンブルしなければならない。

■ファイルSTARLIB.OBJは，STARLIB.ASMだけに依存する。STARLIB.ASMが変更されたら，STALIB.OBJは，指定されたTASMコマンドを使って再アセンブルしなければならない。

このファイルをMAKEFILEという名前でディスクに保存してください。これでMAKE.EXEを使う準備は完了です。

MAKEの使い方

前セクションで説明したソースコードファイルとMAKEFILEがディスクに入っているものとします。MAKEを使うためにしなければならないことは，次のコマンドを入力するだけです。

```
make
```

MAKEは，MAKEFILE（後で説明するように別の名前を付けることもできます）を探して先頭の行を読み込みます。この先頭行には，GETSTARS.EXEの依存関係が記述されています。次にMAKEは，GETSTARS.EXEが存在するかどうか，存在するときは最新バージョンに更新されているかどうか調べます。

GETSTARS.EXEとすべての.OBJファイルが**すでに**存在するときは，MAKEは，各.OBJファイルの作成または更新の日時（日付と時刻）を，そのファイルが依存するファイルの日時と比べます。.OBJファイルが依存するファイルのどれかがその.OBJファイルよりも新しいときは，指定されたTASMコマンドを実行します。

MAKEは，.OBJファイルのどれかを更新したときは，.OBJファイルとGETSTARS.EXE

ファイルの日時を比べます。そして.OBJファイルのどれかがGETSTARS.EXEよりも新しいときは，TLINKコマンドを実行してGETSTARS.EXEを更新します。

GETSTARS.EXEや.OBJファイルが存在しないときは，新しく作成します。

MAKEの動作

先に説明したMAKEの動作を，例を用いて順を追って説明します。いまGETSTARS.EXEとすべての.OBJファイルがすでに存在するとします。またGETSTARS.EXEはどの.OBJファイルよりも新しいとします。同様に各.OBJファイルは，それらが依存するどのファイルよりも新しいものとします。

このとき次のコマンド

```
make
```

を入力しても何も起こりません。なぜなら，どのファイルも更新する必要がないからです。

いまSTARLIB.ASMとSTARLIB.INCを変更したとしましょう。たとえば，どれかの定数の値を変更したとします。こんどは

```
make
```

を入力してMAKEを起動すると，MAKEは，STARLIB.ASMがSTARLIB.OBJよりも新しいことを検知して，コマンド

```
tasm /t /ml /s starlib.asm
```

を実行します。またSTARLIB.INCがGSCOMP.OBJよりも新しいので，コマンド

```
tasm /t /ml /s gscopm.asm
```

を実行します。またSTARLIB.INCがGETSTARS.OBJよりも新しいので，コマンド

```
tasm /t /ml /s getstars.asm
```

を実行します。この3つのコマンドの実行により，STARLIB.OBJ，GSCOMP.OBJ，GETSTARS.OBJはすべて，GETSTARS.EXEよりも新しくなります。そのためMAKEは，次の最終コマンドを実行します。

```
tlink starlib gsparse gscomp gestars, getstars, getstars, lib¥math lib¥io
```

このコマンドは，すべてのファイルをリンクして，GETSTARS.EXEの新しいバージョンを作成します。

これでMAKEについての基本的な事項(主な使用目的, メイクファイルの作成方法, MAKEのメイクファイルの解読の仕方など)は, 理解できたはずです。以下のセクションで, MAKEについてさらに詳しく説明します。

メイクファイルの作成

メイクファイルは, プログラムを更新しなければならないか否かを判断するためのファイルの依存関係と, プログラムを作成するためのコマンドを記述します。メイクファイルは幾つでも作成することができ, 好きな名前を付けることができます。MAKEFILEはメイクファイルのデフォルトのファイル名です。MAKEを起動するときメイクファイル名を指定しないと, MAKEはMAKEFILEを探します。。

メイクファイルは, 任意のASCIIテキストエディタで作成することができます。ルール, 定義, 命令は, すべて改行で終わります。行が長すぎるときは, その行の最後に円記号(¥)を置くことにより, 次の行に続けることができます。

ホワイトスペース（ブランクおよびタブ）は, 隣接する識別子（たとえばファイルの依存関係を示すリストのファイル名）を分離するため, およびルール内のコマンドをインデント（字下げ）するために使用します。

メイクファイルの構成要素

メイクファイルの作成は, 定義, コマンド, 命令などを使ってプログラムを書くことに非常によく似ています。メイクファイルで使用できる構成要素は次のとおりです。

■コメント（注釈）
■明示的ルール
■暗黙のルール
■マクロの定義
■命令――ファイルのインクルード(読み込み), 条件実行, エラーの検出, マクロ定義の解除

次に各構成要素について詳しく説明します。

コメント

コメントは，番号記号（#）で始まります。その行の番号記号#の後の部分は無視されます。コメントはどこにでも置くことができます。またどの桁からはじめてもかまいません。

コメント行の最後に円記号(¥)を付けて，次の行に続けることはできません。コメントが2行以上の長さになるときは，各行に#を使わなければなりません。#の後にある¥は，コメントの一部と見なされます。

次にメイクファイルの中のコメントの例を示します。

```
# GETSTARS.EXEのプロジェクト管理を行なうメイクファイル
getstars.exe: getstars.obj gscomp.obj gsparse.obj starlib.obj
# 次の行の行末，および次の2行の間にはコメントを置くことができない
    tlink starlib gsparse gscomp getstars, getstars, ¥
    getstars, lib¥math lib¥io
# ここにはコメントを置くことができる

getstars.obj: getstars.asm stardefs.inc starlib.inc gscomp.inc gsparse.inc
    tasm /t /ml /s getstars.asm    #ここにはコメントを置くことができる
```

明示的ルール

明示的ルールは，すでによく知っています。前の節で例として示したメイクファイルに使われているからです。明示的ルールの構文規則は次のとおりです。

```
target [target ...]: [source source ...]
      [command]
      [command]
       ...
```

ここで*target*は作成または更新しようとするターゲット（目的とする）ファイル，*source*は*target*が依存するソースファイル，*command*はMS-DOSのコマンド(.BATファイルの呼び出し，.COMファイルや.EXEファイルの実行など）です。

明示的ルールは，ターゲットファイル名（1つまたは複数），ソースファイル（ゼロまたはそれ以上），および実行すべきコマンドのリスト（オプション）を記述します。明示的ルールにリストするターゲットファイルとソースファイルは，MS-DOSのドライブとディレクトリの指定を含んでいてもかまいません。ただしワイルドカードを含むことはできません。

このルールの構文規則は重要です。*target*は行の先頭（1桁目）から始まらなければなりません。また*command*はインデントしなければなりません(少なくとも1つの空白またはタ

ブが必要です)。前に説明したように，ソースファイルのリストやコマンドが長すぎて1行に入りきらないときは，円記号(¥)を継続文字として使って，次の行に続くことを示すことができます。さらにソースファイルとコマンドはどちらも，指定してもしなくてもかまいません。したがって，次のような**ターゲットファイル**とコロンだけからなるルールを書くこともできます。

```
target [target...]:
```

明示的ルールの考え方は，コマンドまたはリストされた複数のコマンドにより，通常は*source*ファイルを使って，*target*ファイルを作成または更新することです。MAKEは，明示的ルールに出会うと，その**ソース**ファイルのどれかが，メイクファイルの別のルールのターゲットファイルになっていないか調べます。ターゲットファイルになっているときは，それらのルールを先に評価します。

ある明示的ルールのすべての**ソース**ファイルのうち，他のルール（明示的ルールまたは暗黙のルール）に基づいて作成または更新する必要のあるファイルの作成や更新が終了したら，MAKEは次に，**ターゲット**ファイルが存在するかどうか調べます。ターゲットファイルが存在しないときは，コマンドリストに指定された**コマンド**を順番に実行します。**ターゲット**ファイルが存在するときは，その最後の更新の日時を，各**ソース**ファイルの日時と比較します。**ソース**ファイルのどれかが**ターゲット**ファイルよりも後に変更されているときは，コマンドを実行します。それ以外のときは，コマンドを実行しません。

1つのファイル名は，1つのメイクファイルの中では，明示的ルールの左側に一度しか現われることができません。

明示的ルール中の各コマンドラインは，空白（またはタブ）で始まらなければなりません。MAKEは，各明示的ルールの下の，次に1桁目から始まる行まで，またはファイルの終わりまでのすべての行を，そのルールのコマンドリストとみなします。先頭の空白は無視します。

明示的ルールに関する注意

コマンドラインが指定されていない明示的ルールは，コマンドラインが指定されている明示的ルールと扱いが少し異なります。

- ■明示的ルールにコマンドが指定されているときは，ターゲットファイルが依存するファイルは，その明示的ルールにリストされているファイルだけです。
- ■明示的ルールにコマンドが指定されていないときは，ターゲットファイルは，その明示的ルールにリストされているファイルと，そのターゲットファイルに適用される暗黙のルールにより決まるファイルに依存します（暗黙のルールについては，後の節を

参照してください)。

明示的ルールの例

```
myprog.obj: myprog.asm
        tasm /t myprog.asm

prog2.obj: prog2.asm include¥stdio.inc
        tasm /t /ml prog2.asm

prog.exe: myprog.asm prog2.asm include¥stdio.inc
        tasm /t myprog.asm
        tasm /t /ml prog2.asm
        tlink myprog prog2, prog,, lib¥io
```

■最初の明示的ルールは，MYPROG.OBJがMYPROG.ASMに依存すること，そしてMYPROG.OBJは，指定されたTASMコマンドを使って作成されることを記述します。

■同様に2番目のルールは，PROG2.OBJがPROG2.ASMとSTDIO.INC(INCLUDEサブディレクトリ内に入っている)に依存すること，そしてTASMコマンドで作成されることを記述します。

■最後のルールは, PROG.EXEがMYPROG.ASM, PROG2.ASM, STDIO.INCの3つのファイルに依存すること, この3つのファイルのどれかが変更されたらその下のコマンドリストにより更新しなければならないこと記述します。
ただしこのやり方では, MYPROG.ASMだけしか変更しない場合にも, PROG2.ASMが再アセンブルされます。これは，ターゲットファイルが依存するファイルよりも旧いと，そのルールの下のすべてのコマンドが実行されるからです。

■メイクファイルの最初の行に，次のような明示的ルールとコマンド

```
prog.exe: myprog.obj prog2.obj
        tlink myprog prog2.obj, prog,, lib¥io
```

を置き，その後に上例のはじめの2つの明示的ルール (MYPROG.OBJとPROG2.OBJのルール) を指定すると，再アセンブルが必要なファイルだけを再アセンブルすることができます。

暗黙のルール

メイクファイルには，暗黙のルールを指定することもできます。暗黙のルールは，明示

的ルールを一般化したものです。次に例を使って，この2種類のルールの関係を説明します。前例の明示的ルール

```
starlib.obj: starlib.asm
        tasm /t /ml /s starlib.asm
```

は一般的なルールです。なぜならこのルールは，.OBJファイルが同じファイル名で拡張子が.ASMのファイルに依存し，TASMコマンドにより作成されるという一般的な慣例にしたがっているからです。実際，これと同じ形式の2，3の（または数十の）明示的ルールを含むメイクファイルを作成することがあるでしょう。

明示的ルールを暗黙のルールで書き換えることにより，同じ形式のすべての明示的ルールが不要になります。上の明示的ルールは，暗黙のルールで書くと次のようになります。

```
.asm.obj:
        tasm /t /ml /s $<
```

このルールは 拡張子が.OBJのすべてのファイルは，拡張子が.ASMの同名のファイルに依存し，コマンド

```
tasm /t /ml /s $<
```

により作成されることを意味します。$<はソース拡張子（.ASM）を持つファイル名を表わします（記号$<は特殊のマクロです。次の節で説明します）。

暗黙のルールの構文規則は，次のとおりです。

```
.source_extension.target_extension:
        [command]
        [command]
         ...
```

明示的ルールの場合と同じように，コマンドは，指定してもしなくてもかまいません。指定するときはインデントしなければなりません。

source_extension（1桁目から始まらなければならない）はソースファイルの拡張子で，次の形式のすべてのファイルを指します。

```
fname.source_extension
```

同じように*target_extension*は，次の形式のファイルを指します。

```
fname.target_extension
```

ここでソースファイルの*fname*とターゲットファイルの*fname*は，同じでなければなりま

せん。言い換えるとこの暗黙のルールは，次の形式のすべての明示的ルールと置き換えることができます。

```
fname.target_extension: fname.source_extension
        [command]
        [command]
         ...
```

MAKEは，コマンドラインで指定されたターゲットファイルを作成するための明示的ルールが見つからないとき，またはコマンドが指定されていない明示的ルールが存在するときに，暗黙のルールを使います。

たとえば，次のような内容のメイクファイルMAKEFILEがあるとしましょう。

```
.asm.obj:
        tasm /t /ml /s $<
```

このときアセンブラプログラムRATIO.ASMをRATIO.OBJにアセンブルしたいとすると，次のコマンドを使うことができます。

```
make ratio.obj
```

MAKEは，RATIO.OBJをターゲットと見なします。そしてRATIO.OBJを作成するための明示的なルールがないので，メイクファイルの暗黙のルールを使用して次のコマンド

```
tasm /t /ml /s ratio.asm
```

を生成し，このコマンドを使ってアセンブルします。

またMAKEは，明示的ルールにコマンドが指定されていないときも，暗黙のルールを使用します。たとえば，前に説明した星の情報を表示するプログラムのメイクファイルの先頭に，次のような暗黙のルールを指定します。

```
.asm.obj:
        tasm /t /ml /s $<
```

そして，.OBJファイルに対する明示的ルールを次のように書き換えます(ファイルの依存関係だけを記述します)。

```
getstars.obj: stardefs.inc starlib.inc gscomp.inc gsparse.inc
gscomp.obj: stardefs.inc starlib.inc
gsparse.obj: stardefs.inc
```

これらの.OBJファイルの作成の方法を指定する明示的ルールがないので，MAKEは，先に指定された暗黙のルールを適用します。STARLIB.OBJはSTARLIB.ASMだけに依存す

るので，このルールは上のリストから除いてあります。指定しなくても自動的に，暗黙のルールが適用されます。

同一のtarget_extensionに対して，複数の暗黙のルールを書くことができます。ただし一時に1つのルールしか適用されません。あるtarget_extensionに対して複数の暗黙のルールが存在するときは，MAKEは，適用可能なルールが見つかるまで，各ルールを順番にチェックします。

MAKEは，ターゲットファイルと同じファイル名で，指定されたsource_extensionを持つファイルが見つかった最初の暗黙のルールを使います。そしてそのルールのコマンドが正常に実行できなくても，それ以上暗黙のルールをチェックしません。

暗黙のルールの下の，次に1桁目から始まる行まで，またはファイルの終わりまでのすべての行を，そのルールのコマンドリストとみなします。各行の先頭の空白は無視されます。コマンドラインの書式は，この章の後の方で説明します。

暗黙のルールに関する注意

MAKEは，暗黙のルールからは完全なファイル名を知ることはできません。そのためMAKEには，特別のマクロが用意されています。このマクロを使うと，暗黙のルール(コマンド)にファイルの名前を読み込むことができます（詳しい説明は，少し後のマクロの説明を参照してください)。

暗黙のルールの例

次に暗黙のルールの例を示します。

```
.c.obj:
     tcc -c $<

.asm.obj:
     tasm $* /mx;
```

最初の例では，ターゲットファイルは.OBJ，それらのソースファイルは.Cファイルです。コマンドリストは1行です。

2番目の例は，TASMに/**MX**オプションを指定して，.ASMソースファイルから.OBJを作成することをMAKEに指示します。

コマンドリスト

次に，明示的ルールおよび暗黙のルールのコマンドリストの指定方法について説明しま

す。

コマンドリストのコマンドは，インデント（字下げ）しなければなりません。すなわち少なくとも1つの空白またはタブで始まらなければなりません。コマンドの書式は次のとおりです。

```
[prefix ...] command_body
```

コマンドリストの各行は，プレフィクスのリスト（指定してもしなくてもよい）とそれに続く1つのコマンドとから構成されます。

プレフィクス

コマンドラインの中に使用できるプレフィクスは，コマンドに関するMAKEの動作を変更します。プレフィクスには，単価記号（@）とハイフン（-）があります。ハイフンには，その後に数を指定することもできます。

@　MAKEがコマンドを実行する前にそのコマンドを表示するのを抑制します。MAKEのコマンドラインに-sオプションを指定していなくても，表示が隠されます。このプレフィクスは，その直後のコマンドに対してのみ有効です。

-num　終了ステータスに対する処理を変更します。数(*num*）を指定すると，MAKEは，終了ステータスが指定した数よりも大きい場合にのみ処理を中止します。下の例では，終了ステータスが4よりも大きい場合に中止します。

```
-4 myprog sample.x
```

-*num*プレフィクスを指定しないときは，MAKEはそのコマンドの終了ステータスをチェックし，終了ステータスが0（正常終了）でなければ，処理を中止してカレントターゲットファイルを削除します。

-　ハイフンだけを指定して数を指定しないときは，MAKEは終了ステータスをチェックしません。終了コードが何であっても，MAKEは処理を続けます。

コマンド

コマンドは，COMMAND.COMの入力行に入力したときとまったく同じように扱われます。ただしリダイレクションとパイプはサポートされません。

MAKEは，COMMAND.COMのコピーを起動することにより，次の内部コマンドを実行

します。

```
break     cd        chdir     cls       copy      ctty
date      del       dir       echo      erase     md
mkdir     path      prompt    rem       ren       rename
set       time      type      ver       verify    vol
```

これ以外のコマンドやバッチファイルは，MS-DOSの検索アルゴリズムを使って探します。

■まずカレントディレクトリを探し，その後パス内の各ディレクトリを探します。
■各ディレクトリの中では，まず拡張子が.COMのファイルを探し，次に.EXEファイルを探し，最後に.BATファイルを探します。
■.BATファイルが見つかったときは，COMMAND.COMのコピーを起動してバッチファイルを実行します。

コマンドラインに拡張子まで指定したときは，MAKEはその拡張子を持つファイルしか探しません。

コマンドリストの例

次の例では，COMMAND.COMによりコマンドが実行されます。

```
cd a:¥include
```

次の例では，検索アルゴリズムを使って全ファイルが探されます。

```
tlink x y,z,z,lib¥io
```

次の例では，拡張子が.COMのファイルだけが探されます。

```
myprog.com geo.xyz
```

次の例では，指定した名前のプログラムが実行されます。

```
a:¥myprog¥file.exe -r
```

マクロ

メイクファイルの中で，いくつかのコマンド，ファイル名，オプションなどが繰り返し出てくることがあります。たとえばこのセクションの最初の例では，TASMコマンドはいつでも，カレントディレクトリの中のソースファイルを探します。いまソースファイルを探すディレクトリと，出力ファイルを入れるディレクトリを制御したいとしましょう。そのためには，メイクファイルの中のディレクトリを変更しなければなりません。マクロを使うと，ディレクトリを1つ1つ変更するかわりに，半自動化することができます。

マクロは文字列を表わす名前です。マクロの定義は，マクロの名前と展開するテキスト定義します。マクロを定義すると，MAKEは，その後そのマクロ名に出会うたびに，その名前をテキストに置き換えます。

メイクファイルの先頭で，次のマクロを定義したとしましょう。

```
SRC = C:¥ASM¥
OUT = OBJS¥
INC = C:¥INC¥
```

そしてメイクファイルのその後の部分が，次のようになっているとしましょう。

```
getstars.exe: $(OUT)getstars.obj $(OUT)gsparse ¥
              $(OUT)gscomp.obj $(OUT)starlib.obj
        tlink $(OUT)starlib $(OUT)gsparse $(OUT)gscomp.obj $(OUT)getstars, ¥
              $(OUT)getstars, $(OUT)getstars, lib¥math lib¥io

getstars.obj: $(SRC)getstars.asm $(INC)stardefs.inc ¥
              $(INC)starlib.inc $(INC)gscomp.inc¥
              $(INC)gsparse.inc
        tasm /t /ml /s /i$(INC) $(SRC)getstars.asm, $(OUT)getstars.obj

gscomp.obj: $(SRC)gscomp.asm $(INC)stardefs.inc $(INC)starlib.inc
        tasm /t /ml /s /i$(INC) $(SRC)gscomp.asm, $(OUT)gscomp.obj

gsparse.obj: $(SRC)gsparse.asm $(INC)stardefs.inc $(INC)starlib.inc
        tasm /t /ml /s /i$(INC) $(SRC)gsparse.asm, $(OUT)gsparse.obj

starlib.obj: $(SRC)starlib.asm
        tasm /t /ml /s /i$(INC) $(SRC)starlib.asm, $(OUT)starlib.obj
```

MAKEを起動すると，$(SRC)はC:¥ASM¥に，$(INC)はC:¥INC¥に，$(OUT)はOBJS¥に置き換えられます。

このようにマクロを使うと，マクロの文字列を変えることにより，そのマクロを含むすべてのコマンドを変更することができます。たとえば，インクルードファイルをC:¥INC STARという名前の新しいサブディレクトリに移したいとしましょう。そのために必要なメイクファイルの変更は，INCマクロを次のように変更することだけです。

```
INC = C:¥INC¥STAR¥
```

これにより，すべてのTASMコマンドを，この新しいインクルードサブディレクトリを使うように変更することができます。

マクロの定義

マクロを定義するための書式は，次のとおりです。

```
macro_name=expansion_text
```

ここで*macro_name*はマクロの名前です。マクロ名は，英字または数字で，空白を含んではいけません。ただし*macro_name*と等号（=）の間には空白を入れてもかまいません。*expansion_text*は，英字，数字，空白，句読記号を含む任意の文字列で，改行すると終わりになります。

*macro_name*が，MAKEのコマンドライン上で**-D**オプションにより定義されているとき，または同じメイクファイルの中で先に定義されているときは，古い定義のかわりに新しい定義が有効になります。

マクロの名前の中では，大文字と小文字は区別されます。したがって**mdl, Mdl, MDL**という名前のマクロは，すべて別のマクロと見なされます。

マクロの使い方

マクロは，メイクファイルの中で，次の書式で呼び出すことができます。

```
$(macro_name)
```

カッコは，マクロの名前が一文字の場合でも必要です。ただしすぐ後で説明する，あらかじめ定義された3つの特殊マクロは例外です。この$(*macro_name*)構造は，マクロ呼び出しと呼ばれます。

MAKEは，マクロ呼び出しに出会うと，その呼び出しをマクロの展開テキストで置き換えます。マクロが定義されていないときは，その呼び出しをヌル文字列で置き換えます。

マクロに関する注意

マクロ中のマクロ

マクロは，マクロ定義の左(*macro_name*)側では呼び出すことができません。マクロ定

義の右（展開テキスト）側では呼び出すことができます。ただし定義されているマクロが呼び出されるまで展開されません。マクロの呼び出しが展開されるときに，そのマクロの展開テキストの中に埋め込まれているマクロも展開されます。

ルール中のマクロ:

ルール行のマクロの呼び出しは直ちに展開されます。

命令中のマクロ:

!ifおよび**!elif**命令の中のマクロの呼び出しは，直ちに展開されます。**!if**または**!elif**命令の中で呼び出されているマクロが定義されていないときは，その呼び出しは，値0(FALSE)に展開されます。

コマンド中のマクロ:

コマンドの中のマクロの呼び出しは，そのコマンドが実行されるときに展開されます。

定義済みマクロ

MAKEには，6つの特殊マクロがあらかじめ定義されています。**$d, $*, $<,$:, $., $&**です。最初のマクロは，条件命令**!if**と**!elif**の中で使用される定義テストマクロです。その他のマクロは，明示的ルールおよび暗黙のルールの中で使用されるファイル名マクロです。またSET環境文字列も，自動的にマクロとしてロードされます。さらにマクロ**__MAKE__**は1と定義されています。

定義テストマクロ（$d）:

定義テストマクロ**$d**は，指定されたマクロが定義されているときは1に，定義されていないときは0に展開されます。マクロの展開テキストの内容は問題にしません。この特殊マクロは，**!if**と**!elif**の2つの命令の中でしか使えません。

たとえば，マクロ**INC**にディレクトリを定義したときはそのディレクトリ，定義しないときはデフォルトのディレクトリのファイルを使用するとしましょう。そのときはこのマクロを使って，メイクファイルの先頭で次のような条件判定をすることができます。

```
!if !$d(INC)       # INCが定義されていないときは
INC=C:¥INC¥        # INCにデフォルトのパスを定義する
!endif
```

このとき，次のようなコマンドライン

```
make -DINC=C:¥INC¥STAR¥
```

でMAKEを起動すると，メイクファイルのINCがC:¥INC¥STAR¥に置換されます。しかし次のコマンドライン

```
make
```

で起動すると，INCはC:¥INC¥（デフォルトのディレクトリ）に置換されます。

ファイル名マクロ:

ファイル名マクロは，どれも，直前に指定された特定のファイルのパス，ファイル名，拡張子の全体またはその一部分に展開されます。

ベースファイル名マクロ（$＊）

ベースファイル名マクロは，明示的ルールまたは暗黙のルールのコマンドの中に使うことができます。このマクロは，ソースファイルのパスとファイル名とに展開されます。

ソースファイル名	**A:¥P¥TESTFILE.ASM**
$*の展開テキスト	**A:¥P¥TESTFILE**

このマクロを使うと，前の例のGETSTARS.EXEの明示的ルールは，次のように書き換えることができます。

```
getstars.exe: getstars.obj gscomp.obj gsparse.obj starlib.obj
        tlink starlib gsparse gscomp $*, $*, $*, ¥
              lib¥emu lib¥math ¥lib¥io
```

このルールのコマンドが実行されるとき，マクロ$＊はファイル名GETSTARSに置き換えられます(拡張子.OBJは除かれます)。このマクロは，暗黙のルールに対して非常に便利です。たとえば暗黙のルールのTASMコマンドは，次のように書くことができます。

```
.asm.obj:
    tasm /t $*
```

フルファイル名マクロ（$<）

このマクロも明示的ルールおよび暗黙のルールに使用します。

明示的ルールの中では，$<はフルターゲットファイル名(拡張子を含む）に展開されます。たとえば

```
ファイル名         A:¥P¥TESTFILE.ASM
$<の展開テキスト   A:¥P¥TESTFILE.ASM
```

したがって次のルール

```
starlib.obj: starlib.asm
      copy $< ¥oldobjs
      tasm /t $*
```

は，まずSTARLIB.OBJをディレクトリ¥OLDOBJSにコピーしてから，STARLIB.ASMをアセンブルします。

暗示ルールの中では，$<は，ファイル名とソース拡張子に展開されます。たとえば先の暗示ルール

```
.obj.asm
    tasm /t $*.asm
```

は，次のように書くこともできます。

```
.obj.asm
    tasm /t $<
```

パスマクロ（$:）

このマクロは，パス（ファイル名は含まない）に展開されます。たとえば

```
ファイル名         A:¥P¥TESTFILE.ASM
$:の展開テキスト   A:¥P¥
```

ファイル名および拡張子マクロ（$.）

このマクロは，拡張子つきのソースファイル名に展開されます。たとえば

```
ソースファイル名   A:¥P¥TESTFILE.ASM
$:の展開テキスト   TESTFILE.ASM
```

ファイル名オンリマクロ（$&）

このマクロは，パスも拡張子も付かないファイル名に展開されます。たとえば

```
ファイル名         A:¥P¥TESTFILE.ASM
$&の展開テキスト   TESTFILE
```

命令

Turbo AssemblerのMAKEでは，他のMAKEと異なり，アセンブラで使用できる疑似命令に似た命令を使用できます。これらの命令を使うと，メイクファイルの中に別のメイクファイルをインクルードしたり，ルールやコマンドの実行に条件を付けたり，エラーメッセージを表示したり，マクロの定義を解除したりすることができます。

アセンブラの命令は番号記号（#）で始まりますが，メイクファイルの中の命令は，行の先頭が感嘆符（!）で始まります。MAKEの命令は次のとおりです。

```
!include
!if
!else
!elif
!endif
!error
!undef
```

ファイルインクルード命令

ファイルインクルード命令（**!include**）は，メイクファイルのその場所にインクルードするファイルを指定します。この命令の書式は次のとおりです。

```
!include "filename"
```

この命令は，ネストすることができます。ネストの深さに制限はありません。この命令により，ネストのあるレベルでインクルードされているファイルをそれよりも内側のレベルでインクルードしようとしたときは，ネストのループを防ぐために，内側の命令はエラーとして拒絶されます。

次にこの命令の使い方について説明します。いま次のようなマクロの定義を含むファイルMODEL.MACを作成したとしましょう。

```
!if $d(MDL)
MDL=medium
!endif
```

次の命令を使うと，この条件マクロ定義をメイクファイルに利用することができます。

```
!include "MODEL.MAC"
```

MAKEは，**!include**命令に出会うと，指定されたファイルをオープンして，その内容をメイクファイルに読み込みます。

条件命令

条件命令（**!if, !elif, !else, !endif**）を使うと，柔軟なメイクファイルを構成することができます。たとえば，ルールの実行やマクロの定義にいろいろな条件を設定して，**-D**オプションによるコマンドライン上のマクロの定義により，メイクファイルのある部分を有効にしたり無効にしたりすることができます。

これらの命令の書式は，アセンブラのプリプロセッサの命令の書式と同じです。

```
!if expression
    [ lines ]
!endif

!if expression
    [ lines ]
!else
    [ lines ]
!endif

!if expression
    [ lines ]
!elsif expression
    [ lines ]
!endif
```

注意：*lines*は，次のどれでもかまいません。

マクロの定義

明示的ルール

暗黙のルール

インクルード命令

ifグループ

エラー命令

定義解除命令

条件命令は，**!if**命令に始まり**!endif**命令により終わるグループを構成します。

■このグループには**!else**命令を1つだけ含むことができます。

■**!if**命令と**!else**命令の間には**!elif**命令を含むことができます。

■条件命令の間には，ルール，マクロ，他の命令などを置くことができます。数に制限はありません。コマンドが指定されている完全ルールは，条件命令で分断することはできません。

■条件命令のグループは，ネストすることができます。ネストの深さに制限はありません。

すべてのルール，コマンド，命令は，そのファイル内で完結していなくてはなりません。 **!if**命令には，同じメイクファイル内に，対応する**!endif**命令がなければなりません。次のようなインクルードファイルは，**!if**に対応する**!endif**がないため，このファイルをインクルードするファイルの内容によらず，構文規則に違反します。

```
!if $(FILE_COUNT) > 5
    <1つまたは複数のルール>
!else
    <1つまたは複数の他のルール>
<ファイルエンド>
```

条件命令の中に使用できる式

!ifまたは**!elif**条件命令の中には，アセンブラの式と同じような式を使うことができます。この式は，32ビットの符号つきの整数式として評価されます。

数は10進数,8進数,または16進数の定数として入力することができます。数の表記方法は，次のとおりです。

```
4536    # 10進数の整数
0677    #  8進数の整数 （先頭に0を付ける）
0x23af  # 16進数の整数 （先頭に0xを付ける）
```

使用できる演算子は次のとおりです。

単項演算子：

-	負符号
~	ビットごとの補数
!	否定

二項演算子

+	加算
-	減算
*	乗算
/	除算
%	余り
>>	右シフト

<<	左シフト
&	ビットごとのAND
|	ビットごとのOR
^	ビットごとの排他OR
&&	論理AND
||	論理OR
>	大きい
<	小さい
>=	大きいかまたは等しい
<=	小さいかまたは等しい
==	等しい
!=	等しくない

条件演算子：

?: 第1オペランド（?の前のオペランド）は，テスト条件として扱われる。第1オペランドの値がゼロでないときは，第2オペランド（?と：の間の部分）が結果になります。第1オペランドの値がゼロのときは，第3オペランド（：の後の部分）が結果になります。

二項演算子は，アセンブラの場合と同じ優先順位にしたがって結合されます。優先順位の等しい演算子に対しては，左から右に結合されます。カッコ(　)を使うと，結合の仕方を変更することができます。条件演算子（?:）は，右から左に結合されます。

式中でマクロを呼び出すこともできます。また特殊マクロ**$d**(　)も使用できます。式中でマクロを呼び出すときは，すべてのマクロが展開された後の式が，構文規則に適合していなければなりません。

エラー命令

エラー命令（**!error**）は，MAKEを停止させ，指定されたメッセージ文を表示します。エラー命令の書式は，次のとおりです。

```
!error <メッセージ文>
```

この命令を条件命令と組み合わせると，ユーザが自分でエラー条件を設定し，そのエラーが発生したときにエラーメッセージを表示させることができます。たとえば，メイクファイルの最初の明示的ルールの前に，次のような命令を挿入します。

```
!if ! $d(MDL)
# MDLが定義されていないとき
!error MDLが定義されていません.
!endif
```

するとMAKEは，マクロ**MDL**が定義されていないときにこの点に到達すると，停止して次のエラーメッセージを表示します。

```
Fatal makefile 5: Error directive: MDLが定義されていません.
```

定義解除命令

定義解除命令（**!undef**）は，マクロの定義を解除します。定義を解除しようとしたマクロが定義されていないときは，何もしません。この命令の書式は次のとおりです。

```
!undef macro_name
```

MAKEの使い方

これまでメイクファイルの作成の仕方について説明しましたので，次にMAKEの使い方について説明します。

コマンドラインの規則

MAKEの最も簡単な使い方は，MS-DOSのコマンドラインに，次のコマンド

```
make
```

を入力することです。するとMAKEは，MAKEFILEを探します。MAKEFILEが見つからないときは，MAKEFILE.MAKを探します。MAKEFILE.MAKも見つからないときは，エラーメッセージを表示して停止します。

MAKEFILEまたはMAKEFILE.MAK以外の名前のメイクファイルを使いたいときは，どうすればいいでしょうか。その場合には，MAKEを起動するときに，次のようにファイルオプション（**-f**）を指定します。

```
make -fstars.mak
```

MAKEの書式を一般的に表わすと，次のようになります。

```
make option option ... target target...
```

ここで*option*はMAKEのオプション（後述）です。*target*は明示的ルールにより処理されるターゲットファイルの名前です。

MAKEのコマンドラインの規則は次のとおりです。

- ■*make*の後に空白を入力し，それからメイクオプションのリストを入力する。
- ■各メイクオプションは，隣接するオプションと空白で隔離しなければならない。オプションは，任意の順番で指定することができる。またコマンドラインに指定できるだけ，いくつでも指定できる。
- ■メイクオプションのリストの後に空白を入力し，それからターゲットファイルのリストを入力する。
- ■各ターゲットファイルは，隣接するターゲットファイルと空白で分離しなければならない。MAKEは，ターゲットファイルを指定された順番に評価する。必要に応じてそれらの構成要素を再アセンブルする。

コマンドラインにターゲットファイル名が指定されていないときは，MAKEは，明示的ルールに指定されている最初のターゲットファイルを使用します。コマンドラインに1つまたは複数のターゲットファイルが指定されているときは，必要に応じてそれらのファイルを作成します。

MAKEのコマンドラインの例をいくつか示します。

```
make -n -fstars.mak
make -s
make -Iinclude -DMDL=compact
```

MAKEを停止させる方法

MAKEは，実行を開始したコマンドがSTOPキー（IBM PCでは***Ctrl-Break***）により中止させられると停止します。また***CTRL-C***も，現在実行中のコマンドを中止させるので，MAKEを停止させます。

BUILTINS.MAKファイル

メイクファイルの中で，同じマクロやルール（通常は暗黙のルール）を繰り返し使用することがあります。また複数のメイクファイルに，同じマクロやルールを使用することもあります。そのようなマクロやルールを扱うのに，3つの方法があります。第1は，作成するすべてのメイクファイルに個別に，それらを入力することです。第2は，それらを1つのファイルに入れておいて，**!include**命令を使ってメイクファイルにインクルードすることです。第3は，それらをBUILTINS.MAKという名前のファイルに入れることです。

BUILTINS.MAKファイルは，複数のファイルで共通に使用されるルール（通常暗黙のルール）やマクロを入れておくためのものです。MAKEは，起動するたびに，BUILTINS.MAKという名前のファイルを探しします。BUILTINS.MAKが見つかったときは，そのファイルを最初に解釈します。BUILTINS.MAKが見つからないときは，MAKEFILE（またはユーザが指定したメイクファイル）の解釈に進みます。

MAKEのメイクファイルの探し方

MAKEは，BUILTINS.MAKを見つけるために，カレントデレクトリまたはパス内のすべてのディレクトリを探します。ただしBUILTINS.MAKファイルは，MAKE.EXEと同じディレクトリに入れるようにしてください。

MAKEがメイクファイルを探すのは，いつでもカレントディレクトリの中だけです。メイクファイルは，実行可能プログラムを作成，更新するためのルールを含みます。BUILTINS.MAKとメイクファイルの構文規則は同じです。

またMAKEは，**インクルード**ファイルを見つけるためにカレントディレクトリを探します。**-I**（インクルード）オプションでディレクトリを指定したときは，指定されたディレクトリの中も探します。

TOUCHユーティリティ

ソースファイルに変更を加えなくても，ターゲットファイルを強制的に再アセンブル（コンパイル）したり，再リンクしたい場合があります。そのための1つの方法は，Turbo Assemblerと一緒に提供されるTOUCHユーティリティを使うことです。TOUCHは，指定された1つまたは複数のファイルの日付と時刻を，現在の日付と時刻に変更します。

ターゲットファイルを強制的に再アセンブルまたは再リンクするには，TOUCHユーティリティを使って，そのターゲットファイルが依存するソースファイルのどれかの日時を新

しくします。TOUCHを使うには，MS-DOSのコマンドラインに

```
touch filename [filename ...]
```

と入力します。するとTOUCHは，指定されたファイルの変更の日時を更新します。ファイルの指定には，DOSのワイルドカード（*，?）を使うこともできます。

ソースファイルのどれかの日時を更新してMAKEを起動すると，MAKEは，ターゲットファイルを再アセンブルまたは再リンクします。

MAKEのコマンドラインオプション

コマンドラインオプションでは，大文字小文字は区別されるので注意してください。次ににすべてのコマンドラインオプションについて説明します。

-a　依存関係のチェックをオンにします。

-D*identifier*　識別子*identifier*を定義します。

-D*iden*=*string*　識別子*iden*を等号の後の文字列*string*の名前として定義します。文字列は，空白やタブを含んではいけません。

-I*directory*　MAKEは，インクルードするファイルを見つけるために，指定されたディレクトリ*directory*を探します（カレントディレクトリも探します）。

-U*identifier*　指定された識別子*identifier*の定義を解除します。

-s　MAKEは，通常，次に実行しようとするコマンドを表示します。このオプションを指定すると，コマンドの表示を抑制することができます。

-n　MAKEが，コマンドを表示するだけで，実行しないようにします。このオプションは，メイクファイルをデバッグするときに便利です。

-f*filename*　ファイル*filename*をメイクファイルとして使います。*filename*が存在しなくて，しかも拡張子が指定されていないときは，そのファイル名で拡張子が.MAKのファイルを探します。

-?または-h　ヘルプメッセージを表示します。

MAKEのメッセージ

MAKEは2種類のメッセージを表示します。致命的エラーメッセージと通常のエラーメッセージです。

致命的エラーが発生したときは，MAKEは，アセンブル（またはコンパイル）を直ちに

停止します。この場合には，エラーの原因を解決してから，MAKEを再起動しなければなりません。

エラーメッセージは，メイクファイルの構文規則違反や内容的な間違いを示します。このエラーが発生したときは，MAKEは，メイクファイルの解釈が終了してから停止します。

致命的エラーメッセージ

Don't know how to make XXXXXXXX

このメッセージは，ルールの中に存在しないファイルがあり，しかもそのファイルを作成するためのルールが存在しないときに表示されます。

error directive: XXXX

このメッセージは，MAKEが，メイクファイルの中で**#error**命令に出会ったときに表示されます。ユーザが !error命令の中に指定したメッセージが，XXXXの位置に表示されます。

Incorrect command-line argument: XXX

このエラーは，MAKEのコマンドラインの指定が正しくないときに表示されます。

Not enough memory

このエラーは，作業領域が一杯になったときに表示されます。このメッセージが表示されたときは，もっと大きなメモリを備えたシステムが必要です。使用しているシステムがすでに640Kのメモリを備えているときは，ソースファイルを簡単にしなければならないでしょう。

Unable to execute command

このメッセージは，コマンドを実行しなければならないときに実行できなかった場合に表示されます。コマンドファイルが見つからなかったとき，またはコマンドのスペルが間違っていたときに表示されます。さらに可能性は小ですが，コマンドファイルの内容が壊れている可能性があります。

Unable to open makefile

このメッセージは，カレントディレクトリにMAKEFILEという名前のファイルがないときに表示されます。

エラーメッセージ

Bad file name format in include statement

!includeの文の中のファイル名は，2重引用符（""）または不等号カッコ（<>）で囲まねばなりません。ファイル名に左側の引用符またはかぎカッコがありません。

Bad undef statement syntax

!undef文は，識別子を1つだけ含み，それ以外のものを含んではいけません。

Character constant too long

文字定数は，1文字または2文字でなければなりません。

Command arguments too long

MAKEにより実行されるコマンドに指定した引数が，127文字(MS-DOSにおける上限)以上でした。

Command syntax error

このメッセージは，次のどれかの場合に表示されます。

■メイクファイルの最初のルール行の先頭に空白が入っていた。
■暗黙のルールが.ext.extから構成されていなかった。
■明示的ルールのコロン（:）の前にファイル名が指定されていなかった。
■マクロの定義の等号（=）の前に名前が指定されていなかった。

Division by zero

!if文の中の商または余りを求める式の除数がゼロになります。

Expression syntax error in !if statement

!if文の中の式が構文規則に違反しています。左右のカッコの数が合わない，必要な演算子が抜けている，不必要な演算子が入っている，その他いろいろな原因が考えられます。

File name too long

!include文の中に指定されたファイル名（ドライブ名，パスを含む）が長すぎます。MS-DOSでは，コマンドラインに指定できるファイル名は，最大64文字です。

Illegal character in constant expression

MAKEが，定数式の中で使用することが許されていない文字に出会いました。その文字が英字のときは，その定数式はスペルの間違った識別子である可能性があります。

Illegal octal digit

8進数の定数に，数字の8または9が含まれていました。

Macro expansion too long

マクロは，展開された長さが4,096文字を越えることはできません。このエラーは，しばしば，マクロが自分自身を再帰的に展開する場合に発生します。

Misplaced elif statement

対応する**!if**命令のない**!elif**命令に出会いました。

Misplaced endif statement

!if命令に出会わないのに，**!endif**命令に出会いました。

No file name ending

!include文の中のファイル名に，右側の引用符（"）または不等号カッコ（>）がありませんでした。

Redefinition of target XXXXXXXX

複数の明示的ルールの左側の部分（ターゲットファイル）に，同じファイル名が指定されています。

Unable to open include file XXXXXXXXXXXX

!include文の中に指定されたインクルードファイルが見つかりませんでした。このエラーは，インクルードファイルがそれ自身をインクルードしているときにも発生することがあります。指定したインクルードファイルが存在するかどうかチェックしてください。

Unexpected end of file in conditional started on line #

MAKEが**!endif**に出会う前にメイクファイルの終わりに到達しました。**!endif**が脱落しているか，綴りが間違っているかどちらかです。

Unknown preprocessor statement

行の先頭が感嘆符!であるのに，その後に続く文が**error, undef, if, elif, include, else,** または**endif**ではありませんでした。

TLINK

TLINK（Turbo Linker）は，アセンブラとは別個のプログラムとして起動されます。したがってスタンドアロンのリンカとしても使用することができます。

TLINKには他のリンカの持つ（さほど重要でない）細かな機能はありませんが，そのかわりコードサイズが非常に小さく非常に高速です。

Turbo Assemblerは，アセンブルが成功するとTLINKを起動するようになっています。TLINKは,起動されるとオブジェクトモジュールやライブラリファイルを連結して実行可能ファイルを作成します。

この付録では，TLINKのスタンドアロンリンカとしての使い方を説明します。

TLINKの起動

DOSのコマンドラインに**tlink**と入力することにより，TLINKを起動することができます。起動するときにパラメータを指定することもできます。

パラメータを指定しないで起動すると，TLINKは，次のようにパラメータとオプションの一覧を表示します。

```
Turbo Link  Version 2.0  Copyright (c) 1987, 1988 Borland International
Syntax:  TLINK objfiles, exefile, mapfile, libfiles
@xxxx indicates use response file xxxx
Options: /m = map file with publics
         /x = no map file at all
         /i = initialize all segments
         /l = include source line numbers
         /s = detailed map of segments
         /n = no default libraries
         /d = warn if duplicate symbols in libraries
         /c = lower case significant in symbols
         /3 = enable 32-bit processing
         /v = include full symbolic debug information
         /e = ignore Extended Dictionary
         /t = create COM file
```

上の表示の中の行

```
Syntax:  TLINK objfiles, exefile, mapfile, libfiles
```

は，指示された種類のファイルを，**指示された順番に**，カンマで区切りながら指定しなけ

ればならないことを示します。

たとえば次のコマンドライン

```
tlink /c mainline wd ln tx, fin, mfin, lib¥comm lib¥support
```

を入力したとき，TLINKはこれを次のように解釈します。

■リンク時に文字の大小を区別する（/c）。
■リンクする.OBJファイルはMAINLINE.OBJ，WD.OBJ，LN.OBJおよびTX.OBJである。
■作成する実行可能ファイルの名前はFIN.EXEである。
■マップファイルの名前はMFIN.MAPである。
■リンクするライブラリファイルはCOMM.LIBとSUPPORT.LIBである。どちらのライブラリファイルもサブディレクトリLIBの中にある。

TLINKは，拡張子の指定されていないファイルには拡張子を付け加えます。付け加える拡張子は次のとおりです。

■オブジェクトファイルには.OBJ
■実行可能ファイルには.EXE。/**t**オプションを指定して作成された実行可能ファイルには.COM。
■マップファイルには.MAP
■ライブラリファイルには.LIB

実行可能ファイルの名前が指定されていないときは，TLINKは，リストの最初のオブジェクトファイルの名前に拡張子.EXEを付けることにより，実行可能ファイルの名前を生成します。たとえば，前のコマンドラインの例で実行可能ファイルの名前FINを指定しなければ，TLINKは，実行可能ファイルとしてMAINLINE.EXEを作成します。

TLINKは，コマンドライン上に/**x**オプションを指定してマップファイルを作成しないようにしたとき以外は，いつでもマップファイルを作成します。

■/**m**オプションを指定すると，マップファイルはパブリックシンボルのリストになります。
■/**s**オプションを指定すると，マップファイルは詳細なセグメントマップになります。
TLINKは，次の規則にしたがってマップファイルの名前を決めます。
■.MAPファイルが指定されていないときは，.EXEファイルの名前に拡張子.MAPを付け

ることにより，マップファイルの名前を生成します(.EXEファイルの名前は，コマンドライン上またはレスポンスファイルの中に指定できます)。.EXEファイル名が指定されていないときは，最初の.OBJファイルの名前を使って 作成します。

■コマンドライン上またはレスポンスファイルの中に.MAPファイルの名前だけが指定されているときは，その名前に拡張子.MAPを付加します。

マップファイル名を指定していても，/**x**オプションを指定しているとマップファイルは作成されません。

レスポンスファイルの使い方

TLINKのパラメータは，コマンドラインに入力するか，レスポンスファイルに設定するか，またはその2つの組合わせにより指定することができます。

レスポンスファイルは，通常コマンドライン上の“TLINK”の後に入力するオプションやファイル名を含むテキストファイルです。

ただしコマンドラインに入力する場合と異なり，レスポンスファイルでは，複数行にわたってもかまいません。オブジェクトファイルやライブラリファイルのリストは，複数の行に分割し，各行の末尾にプラス記号（＋）を置くことにより，次の行に続けることができます。

また4つの要素（オブジェクトファイル,実行可能ファイル,マップファイル,ライブラリファイル）を，別々の行に書くこともできます。その場合には，各要素（ファイルの種類）を分離するためのカンマは取り除かなければなりません。

レスポンスファイルの機能を説明するために，先のコマンドラインの例を，次のようなレスポンスファイルに書き直します。このレスポンスファイルの名前をFINRESPとします。

```
/C mainline wd+
ln tx, fin+
mfin+
lib¥comm lib¥support
```

この場合には，TLINKコマンドを次のように入力します。

```
tlink @finresp
```

レスポンスファイル名には，それがレスポンスファイルであることを示すために，頭に単価記号（@）をつけなければならないことに注意してください。

またコマンドラインの全部または一部を，複数のレスポンスファイルに分割して入れる

こともできます。たとえば，上例のコマンドラインのオブジェクトファイルとライブラリファイルを，2つのレスポンスファイルに入れます。

```
ファイル名        内容
LISTOBJS        mainline+
                wd+
                ln tx
LISTLIBS        lib¥comm+
                lib¥support
```

この場合には，TLINKコマンドは次のように入力します。

```
tlink /c @listobjs, fin, mfin, @listlibs
```

TLINKオプション

TLINKオプションは，コマンドラインのどこにでも入力できます。オプションは，スラッシュ（/）とそれに続くオプション文字（***m,s,l,i,n,d,x,3,v,e,t,c***）で指定します。

複数のオプションを指定するとき，オプションの間に空白を入れなくてもかまいません（**/m/c**は**/m /c**と同じです）。またコマンドラインのいろいろな場所に入力することができます。

/x, /m, /sオプション

TLINKは，何も指定しなければ，実行可能ファイルのデフォルトのマップを作成します。このマップファイルには，プログラムのセグメントのリスト，プログラムの開始アドレス，リンク中に作成された警告メッセージやエラーメッセージが入れられます。

マップファイル作成したくないときは**/x**を使います。**/x**を指定すると，オプションによりマップファイルの作成が指定されている場合にも，マップファイルの作成を抑制することができます。

/mオプションを指定すると，デフォルトのマップファイルにパブリックシンボルのリストが付加されます。パブリックシンボルのリストには，パブリックシンボルが下位アドレスのものから順番に列挙されます。このマップファイルは，デバッグに便利です。Periscopeをはじめとする多くのデバッガは，このパブリックシンボルのリストを利用することができます。したがってそれらのデバッガでも，デバッグ中にシンボリックアドレスの参照ができます。

/sオプションを指定すると，**/m**オプションにより作成されるマップファイルの情報に，さ

らに詳細セグメントマップが付加されたマップファイルが作成されます。次に詳細セグメントマップの一例を示します。

Address	Length (Bytes)	Class	Segment Name	Group	Module	Alignment/ Combining
0000:0000	0E5B	C=CODE	S=SYMB_TEXT	G=(none)	M=SYMB.ASM	ACBP=28
00E5:000B	2735	C=CODE	S=QUAL_TEXT	G=(none)	M=QUAL.ASM	ACBP=28
0359:0000	002B	C=CODE	S=SCOPY_TEXT	G=(none)	M=SCOPY	ACBP=28
035B:000B	003A	C=CODE	S=LRSH_TEXT	G=(none)	M=LRSH	ACBP=20
035F:0005	0083	C=CODE	S=PADA_TEXT	G=(none)	M=PADA	ACBP=20
0367:0008	005B	C=CODE	S=PADD_TEXT	G=(none)	M=PADD	ACBP=20
036D:0003	0025	C=CODE	S=PSBP_TEXT	G=(none)	M=PSBP	ACBP=20
036F:0008	05CE	C=CODE	S=BRK_TEXT	G=(none)	M=BRK	ACBP=28
03CC:0006	066F	C=CODE	S=FLOAT_TEXT	G=(none)	M=FLOAT	ACBP=20
0433:0006	000B	C=DATA	S=_DATA	G=DGROUP	M=SYMB.ASM	ACBP=48
0433:0012	00D3	C=DATA	S=_DATA	G=DGROUP	M=QUAL.ASM	ACBP=48
0433:00E6	000E	C=DATA	S=_DATA	G=DGROUP	M=BRK	ACBP=48
0442:0004	0004	C=BSS	S=_BSS	G=DGROUP	M=SYMB.ASM	ACBP=48
0442:0008	0002	C=BSS	S=_BSS	G=DGROUP	M=QUAL.ASM	ACBP=48
0442:000A	000E	C=BSS	S=_BSS	G=DGROUP	M=BRK	ACBP=48

このマップは，各モジュールの各セグメントのアドレス，バイト長，クラス，セグメント名，グループ，モジュール名，ACBP情報を示します。

1つのモジュールが複数のセグメントを含むときは，そのモジュール名が複数の行に現れます（たとえばSYMB.ASM）。詳細セグメントマップの中の情報は，*ACBP*フィールドを除いて，説明の必要はありません。

*ACBP*フィールドは，インテル社の定義にしたがって，A属性（**アラインメント**）とC属性（**異なるモジュールの同名のセグメントの結合方法**）を，1組の4ビットフィールドに符号化します。TLINKは，それらのフィールドのうちの2つのフィールド，*A*フィールドと*C*フィールドだけを使用します。マップの*ACBP*値は，次の表に示す*A*フィールドと*C*フィールドの値をOR演算した値（16進数）です。

フィールド	値	説明
Aフィールド	00	絶対セグメント
(アラインメント)	20	バイトアラインドセグメント
	40	ワードアラインドセグメント
	60	パラグラフアラインドセグメント
	80	ページアラインドセグメント
	A0	メモリの名前のつけられていない絶対部
Cフィールド	00	結合されない
(結合)	08	パブリック結合セグメント

/lオプション

このオプションを指定すると，マップファイルの中にソースコードの行番号が付加されます。このオプションを使うには，行番号オプションをオンにして.OBJファイルを作成しておかなければなりません。/xオプションを指定しているときは，このオプションを指定しても無効です。

/iオプション

このオプションを指定すると，後続のセグメントがデータレコードを含まない場合にも，実行可能ファイルに出力されます。このオプションは，通常は必要ありません。

/nオプション

このオプションを指定すると，TLINKは，いくつかのコンパイラにより指定されるデフォルトのライブラリを無視します。このオプションは，デフォルトのライブラリが別のディレクトリにあるときに必要です。なぜならTLINKには，ライブラリを探す機能がないからです。異なる言語で書かれたモジュールをリンクするときも，このオプションを使いたい場合があるでしょう。

/cオプション

このオプションを指定すると，パブリックシンボルおよび外部シンボルの文字の大小が区別されます。/cオプションを指定しないときは，TLINKは，たとえば*fred*, *Fred*, *FRED*を同等と見なし区別しません。/cオプションを指定したときは，これらを区別します。

/dオプション

TLINKは，通常1つのシンボル（ルーチンの名前）が複数のライブラリファイルに重複して現われても警告しません。そして複数のライブラリファイルに重複して現われるシンボルをプログラムにインクルードしなければならないときは，コマンドラインに指定された最初のライブラリファイルのシンボルを使います。これはよく使用される機能なので，TLINKは通常，シンボルの重複について警告しません。この機能は次のように利用します。

いま2つのライブラリSUPPORT.LIBとDEBUGSUP.LIBがあるとしましょう。そしてDEBUGSUP.LIBは，SUPPORT.LIBの中のいくつかのルーチンのコピーを含むとします(ただしDEBUGSUP.LIBの中のコピー ルーチンはデバッグのためのバージョンで，SUPPORT.LIBのルーチンとは機能が少し異なるとします)。このときTLINKのコマンドライン上でDEBUGSUP.LIBを先に指定すると，SUPPORT.LIBの中のルーチンではなくDEBUGSUP.LIBの中のデバッグルーチンがリンクされます。

この機能を使わないとき，またはどのルーチンが重複しているかわからないときは，/dオプションを使うことができます。このオプションを指定すると，TLINKは，複数のライブラリに重複して存在するすべてのシンボル（ルーチン名）をリストします。プログラムにリンクされないシンボルもリストします。

また/dオプションを指定すると，TLINKは.OBJファイルと.LIBファイルの両方に現われるシンボルについても警告します。コマンドラインにリストされた最初の（最も左側の）ファイルに現われるシンボルがリンクされますので，この場合には.OBJファイルの中のシンボルが使用されます。

/eオプション

Turbo Cと一緒に提供される複数のライブラリファイルには，それぞれ拡張辞書(Extended Dictionary)が含まれています。拡張辞書の情報により，TLINKは，それらのライブラリのルーチンをより高速にリンクすることができます。この拡張辞書は，TLIBユーティリテ

ィに/eオプションを指定することにより，他のどのライブラリファイルにも付け加えることができます（TLIBについては次のセクションを参照してください）。

拡張辞書を含むライブラリのルーチンのリンクは，そうでないライブラリのルーチンのリンクに比べて高速ですが，次の2つの理由により拡張辞書の使用を禁止する/eオプションを使いたいことがあります。

■拡張辞書を使うときは，リンクのために少し大きなメモリが必要である。
■/eを指定しないときは，TLINKは，拡張辞書が付加されたライブラリに含まれるデバッグ情報を無視する。

/tオプション

ソースファイルをタイニィメモリモデルでコンパイルし，このオプションを指定してリンクすると，TLINKは，通常の.EXEファイルのかわりに.COMファイルを作成します。

/tを指定したときは，実行可能ファイルのデフォルトの拡張子は.COMです。

注意：.COMファイルは，サイズが64Kを越えることはできません。セグメントに関する設定はできません。スタックセグメントを定めることもできません。開始アドレスは0:100hでなければなりません。.COM以外の拡張子（たとえば.BIN）が付けられている実行可能ファイルは，開始アドレスが0:0または0:100hのどちらかです。

/vオプション

このオプションは，TLINKに，実行可能ファイルにデバッグ情報を含めるように指示します。

/3オプション

リンクされる1つまたは複数のオブジェクトモジュールがTASMまたはその互換アセンブラにより作成され，80386の32ビットコードを含むときに使います。このオプションを指定すると，TLINKが必要とするメモリが大きくなり，リンクの速度が低下しますので，必要なときだけ使用してください。

制約

先に説明したように，TLINKは，コードサイズが非常に小さく非常に高速です。そのためTLINKには最小限のオプションしかありませんが，重大な制約は次のものだけです。

■オーバーレイがサポートされない。
■共通変数が部分的にしかサポートされない。共通変数を解決するには，パブリック指定しなければならない。
■論理セグメントの数は最大約4,000である。
■同名，同クラスのセグメントは，すべて結合できるかできないかでなければならない。
■Microsoft CおよびMicorosoft Fortranでコンパイルされたコードは，リンクできない。Microsoftの言語コンパイラで作成される.OBJファイルは，TLINKが現在サポートしていないオブジェクトレコードフォーマットを含むからである。

TLINKは，Turbo Assembler, Turbo C(統合環境版およびコマンドライン版の両方)，Turbo Prolog およびその他のコンパイラとともに使用するように設計されています。したがっていつでもMS-Linkのかわりに使用できるとはかぎりません。

メッセージ

TLINKは，3種類のメッセージを表示します。致命的エラーメッセージ，非致命的エラーメッセージ，警告メッセージです。

■致命的エラーを検出すると，ただちに停止して，致命的エラーメッセージを表示します。.EXEファイルと.MAPファイルは削除します。
■非致命的エラーを検出すると，非致命的エラーメッセージを表示します。.EXEファイルも.MAPファイルも削除しません。ただし.EXEファイルを実行してはいけません。
■ユーザが修正したいと思うであろう箇所を検出すると，警告メッセージを表示します。警告箇所を検出したときも.EXEファイルと.MAPファイルを作成します。

この後に掲載されているメッセージには，シンボル名，モジュール名，ファイル名および数値は，それぞれ次の書式で表示されます。

<sname>	シンボル名
<mname>	モジュール名
<fname>	ファイル名
xxxxh	4桁の16進数，最後に'h'を付ける．

致命的エラーメッセージ

致命的エラーが検出されたときは，TLINKは，リンクを停止して.EXEファイルと.MAPファイルを削除します。

XXXXXXXX.XXX: bad objext file

不正な形式のオブジェクトファイルに出会いました。このエラーの原因で最も多いのは，間違えてソースファイルの名前を指定することと，指定したオブジェクトファイルが不完全であることです。不完全なオブジェクトファイルは，アセンブル中にシステムをリブートしたときや，***STOP*** (***Ctrl-Break***) を押したときにアセンブラ（またはコンパイラ）が出力オブジェクトファイルを削除しなかったときに作成されます。

XXXXXXXX.XXX: unable to open file

このエラーは，指定されたファイルが存在しないとき，または名前が間違っているときに発生します。

Bad character in parameters

このエラーは，コマンドラインまたはレスポンスファイルの中で

" * < = > ? [ ] |

のような記号や，水平タブ，改行，復帰，***CTRL-Z***以外の制御文字に出会ったときに発生します。

msdoserror, ax=XXXXh

このエラーは，MS-DOSコールが予期しないエラーを返したときに発生します。表示されたAXレジスタの値XXXXhが，エラーコードを示します。このエラーメッセージは，TLINKの内部エラーまたはMS-DOSのエラーを示します。TLINKの行なうMS-DOSコールでこのエラーが発生する可能性があるのは，ファイルの読み込み，書き込み，クローズです。

Not enough memory

メモリ不足のためリンク作業を完了することができませんでした。メモリに常駐している不要なプログラムを取り除いたり，RAMディスクのサイズを縮小したりして，使用可能なスペースを拡大してから，もう一度TLINKを起動してください。

Segment exceeds 64K

このメッセージは，複数のモジュールの同じ名前のセグメントが結合されたとき，その合計サイズが64Kbを越えた場合に表示されます。また複数のセグメントがGROUP命令により結合されたとき，そのグループの合計サイズが64Kbを越えた場合にも表示されます。

Symbol limit exceeded

1リンクで扱うことができるパブリックシンボル，セグメント名，グループ名の総数は，最大8,182個です。このメッセージは，その制限を越えたときに表示されます。

Unexpected group definition

オブジェクトファイルの中のグループの定義は，特定の順序で現われなければなりません。このメッセージは，アセンブラ（コンパイラ）により欠陥のあるオブジェクトファイルが作成されたときにだけ表示されます。Turbo Assemblerで作成したファイルにこのエラーが検出されたときは，そのファイルを再アセンブルしてください。それでも問題が解決しないときは，MSAカスタマーサポートセンターに連絡してください。

Unexpected segment definition

オブジェクトファイルの中のセグメントの定義は，特定の順序で現われなければなりません。このメッセージは，コンパイラ（またはアセンブラ）により欠陥のあるオブジェクトファイルが作成されたときにだけ表示されます。Turbo Assemblerで作成したファイルにこのエラーが検出されたときは，そのファイルを再アセンブルしてください。それでも問題が解決しないときは，MSAカスタマーサポートセンターに連絡してください。

Unknown option

コマンドライン上またはレスポンスファイルの中で，スラッシュ（/）の後に何も指定されていないか，または不正な文字が指定されていました。

Write failed, disk full?

このエラーは，TLINKが，ディスクに書き込もうとしたすべてのデータを書き込めなかったときに発生します。ほとんどの場合，ディスクが一杯になっています。

非致命的エラーメッセージ

TLINKが検出する非致命的エラーは2つだけです。すでに説明したように，TLINKは非致命的エラーを検出したときは.EXEと.OBJを削除しません。非致命的エラーのエラーメッセージは次のとおりです。

XXX is unresolved in module YYY

モジュールYYYの中で参照されているシンボルXXXは，リンクされるオブジェクトファイルやライブラリルーチンのどれにも定義されていません。シンボルのスペルが間違っていないかどうか調べてください。

Fixup overflow, frame = xxxxh, target = xxxxh,
offset = xxxxh in module XXXXXXXX

このメッセージは，TLINKがリンク時に解決しなければならないオブジェクトファイル中のデータ参照またはコード参照で，正しくないものを示します。オブジェクトファイルは，参照を解決するための付加情報の中に，参照されるメモリロケーションの名前と，そのメモリロケーションが存在するはずのセグメントの名前を示します。*frame*値は，オブジェクトファイルに示されている，メモリロケーションが存在するはずのセグメントです。*target*値は，そのメモリロケーションが実際に存在するセグメントです。*offset*は，ターゲットセグメント内の，そのメモリロケーションのオフセットです。

このメッセージは，しばしば，メモリモデルのミスマッチにより引き起こされます。最もよくある原因は，異なるコードセグメント内の関数を**near**コールしていることです。またデータ変数を**near**コールしたときや，関数をデータ参照したときも，このエラーが発生します。

原因を調べるには，パブリックシンボルのマップを作成してください(**/m**)。エラーメッセージ内のターゲット値とオフセットは，参照されているシンボルのアドレスにならなければなりません。もしターゲットとオフセットがマップのシンボルとマッチしないときは，アドレスが最も近いシンボルを探してください。この参照は表示されたモジュール内にありますので，そのモジュールのソースファイルの中の不正な参照を探してください。 この手法でエラーの原因を突き止めることができないとき，またはTurbo Assembler以外の言語（他社のアセンブリ言語や高級言語）でプログラミングしているときは，他の原因の可能性もあります。Turbo Assemblerでプログラミングしているときでも，使用しているメモリモデルのデフォルトのセグメント名やグループ名と異なるセグメント名やグループ名を使っていると，このメッセージが表示されることがあります。

警告メッセージ

TLINKの出す警告は次の3つだけです。最初の2つは，シンボルの定義の重複に関するものです。3つ目は，スタックが定義されていないことを示します。

Warning:XXX is duplicated in module YYY

シンボル**XXX**は，モジュール**YYY**の中で2度定義されています。このメッセージは，たとえば文字の大小を区別してアセンブルされたオブジェクトモジュールを，文字の大小を区別せずにリンクしようとした場合に表示されます。

Warning:XXX defined in module YYY is duplicated in module ZZZ

モジュール**YYY**で定義されているシンボル**XXX**が，モジュール**ZZZ**でも重複して定義されています。この警告は，コマンドライン上に同じオブジェクトファイルを複数回指定したときなどに出されます。

Warning: no stack

このメッセージは，リンクされるオブジェクトファイルやライブラリルーチンのどれにもスタックセグメントが定義されていない場合に表示されます。Turbo Cのタイニィモデルのプログラムや.COMファイルに変換されるプログラムのときは，このメッセージが表示されても問題はありません。その他のプログラムに対しては警告です。

タイニィメモリモデル以外のTurbo Assemblerのプログラムに対してこのメッセージが表示されたときは，C0*x*スタートアプオブジェクトファイルに間違いがないかチェックしてください。

TLIB

TLIB（ターボライブラリアン）は，.OBJ（オブジェクトモジュール）ファイルのライブラリを管理するためのユーティリティです。ライブラリファイルは，オブジェクトモジュールの集合を単一のファイルとして扱うことができる非常に便利な方法です。

TLIBを使うと，自分自身のライブラリを作成することができます。また自分のライブラリや他のユーザの作成したライブラリ，購入したライブラリなどを変更することができます。TLIBを使うと，次のことができます。

■複数のオブジェクトモジュールから新しいライブラリを作成する。
■ライブラリにオブジェクトモジュールや別のライブラリを付け加える。
■ライブラリからオブジェクトモジュールを削除する。
■ライブラリのオブジェクトファイルを入れ替える。
■ライブラリからオブジェクトモジュールを読み出す。
■ライブラリの内容をリストする。

既存のライブラリを変更するときはいつでも，TLIBは，元のライブラリのコピーを作成し，拡張子.BAKを付けます。

またTLIBは，ライブラリの拡張辞書を作成し，それをそのライブラリに組み込むこともできます。ライブラリに拡張辞書を付加すると，リンクをスピードアップすることができます。詳しい説明は **/E**オプションの項を参照してください。

TLIBは，Turbo Assemblerを使って実行可能ファイルを作成するために必要不可欠なものではありませんが，プログラミングの効率の向上に役立つツールです。大きな開発プロジェクトには，なくてはならないものであることがわかるでしょう。

オブジェクトモジュールライブラリを使う利点

アセンブリ言語でプログラムを書くときは，多くの場合，アセンブラのランタイム（実行時）ライブラリの中の関数のような便利なアセンブラ関数を多数作成します。いまモジュール構造化するために，それらの関数を別々にアセンブルされる複数のソースファイルに分割するとします。そしてそれらのソースファイルの集合の中から，いくつかの関数だけをプログラムに組み入れるとしましょう。しかしこの方法では，ファイルを調べてそれ

を組み入れることが非常にやっかいになります。その反対に，いつでもすべてのソースファイルをインクルードすると，プログラムは非常に大きく膨れ上がります。

オブジェクトモジュールライブラリは，アセンブラ関数の集合を管理する困難を解決します。ユーザのプログラムをライブラリとリンクするときは，リンカは，オブジェクトモジュールライブラリを検索して，自動的にそのプログラムに必要なモジュールを選択します。さらにライブラリは，オブジェクトモジュールファイルの単なる集積に比べて，ディスクスペースが少なくてすみます。各オブジェクトモジュールのサイズが小さくて数が多い場合には特にそうです。またライブラリは，リンク作業をスピードアップします。なぜなら，各オブジェクトモジュールのファイルをオープンするかわりに，1つのライブラリだけをオープンするだけでよいからです。

TLIBのコマンドラインの要素

TLIBのコマンドラインの書式は，次のとおりです。カッコ()で囲んだ項目は，指定してもしなくてもかまいません。

```
tlib libname [/C] [/E] [operations] [, listfile]
```

この節では，コマンドラインの要素について説明します。そして次の節でTLIBの使い方について詳しく説明します。

要素	説明
tlib	TLIBを起動するコマンドです。
libname	作成または管理しようとするライブラリの名前（ドライブ名，パスを含む）です。TLIBを起動するときはいつでも，**ライブラリ名を**指定しなければなりません。ライブラリ名に拡張子を指定しないときは.LIBであると見なされます。ライブラリ名には.LIB以外の拡張子を使わないようにしてください。TASMのMAKEは，拡張子が.LIBのファイルだけしかライブラリファイルと見なさないからです。 add操作が指定されていて，指定されたライブラリが存在しないときは，TLIBは新しいライブラリを作成します。
/C	「文字の大小を区別する」ようにするフラグです。このオプションは通常は使いません。詳しい説明は「文字の大小の区別」/Cオプション を参照してください。
/E	拡張辞書を作成します。詳しい説明は「拡張辞書の作成」を参照してください。
operations	TLIBに行なわせる操作のリストです。操作(operation)は，どのような順番で指定してもかまいません。ライブラリの内容を調べるだけのときは，何も指定しなくてもかまいません。
listfile	ライブラリの内容のリストを出力するファイルの名前です。このファイル名を指定するときは，その前にカンマを入力しなければなりません。ファイル名を指定しないときは，リストファイルは作成されません。このリストは，モジュール名をアルファベット順に並べ，各モジュール名の下にそのモジュールで定義されているパブリックシンボルをアルファベット順に並べたものです。 ファイル名としてCONを指定すると，リストを画面に表示することができます。またPRNを指定すると，プリンタで印刷することができます。

操作リスト

操作リストは，TLIBに行なわせたいこと（アクション）を順番に並べたものです。各操作は，1文字または2文字のアクション記号と，それに続くファイル名またはモジュール名とにより指定します。アクション記号とファイル名またはモジュール名の前後には，空白を入力してもかまいません。ただし2文字のアクション記号の間や名前の途中に，空白を入れることはできません。

操作は，DOSのコマンドラインの長さの制限（127文字）を越えない限り，コマンドライン上にいくつでも指定できます。操作の順番は重要ではありません。TLIBはいつでも，次の順番で操作を実行します。

1. 最初にすべての抽出操作を行なう。
2. 次にすべての削除操作を行なう。
3. 最後にすべての追加操作を行なう。

モジュールの置換は，古いモジュールの削除と，新しいモジュールの追加の2つの基本操作の組み合わせにより行なわれます。

ファイル名とモジュール名

TLIBによりライブラリに追加するオブジェクトモジュールファイルを，単に**モジュール**と呼びます。TLIBは，指定されたファイル名を取り込み，そのファイル名からドライブ名，パス，拡張子の情報を取り除いて，モジュールの名前を見つけ出します（ただしドライブ名，パス，拡張子は，通常は指定しません）。

TLIBは，拡張子やディレクトリの指定がないときは，デフォルトを使います。たとえば，拡張子.OBJを持つモジュールをカレントディレクトリからライブラリに追加するには，モジュール名を指定するだけですみます。パスと拡張子.OBJは，指定しなくてもかまいません。

ファイル名すなわちモジュール名には，ワイルドカードは使用できません。

TLIBの操作

TLIBは，3つのアクション記号（-,+,*）を認識します。これらのアクション記号は，単独でまたは2つの記号を組み合わせて使うことができ，合計で5つの異なる操作を指定す

ることができます。記号の組み合わせで指定する操作にたいしても，記号の順番は重要ではありません。アクション記号とそれにより指定される操作は次のとおりです。

アクション記号	名前	説明
+	追加	TLIBは，指定されたライブラリに指定されたモジュールを追加します。オブジェクトファイルに拡張子が指定されていないときは.OBJであると見なします。指定されたファイルがそれ自体ライブラリ（拡張子.LIB）であるときは，そのライブラリのすべてのモジュールを，追加先のライブラリに追加します。追加されるモジュールと同名のモジュールがすでに存在するときは，メッセージを表示します。モジュールは追加しません。
−	削除	TLIBは，指定されたライブラリから指定されたモジュールを削除します。指定されたモジュールが指定されたライブラリに存在しないときは，メッセージが表示されます。
*	抽出	TLIBは，指定されたライブラリの指定されたモジュールを，指定されたファイルにコピーします。指定されたファイルが存在しないときは，新たに作成します。指定されたファイルが存在するときは，古い内容に上書きします。指定されたモジュールが存在しないときは，メッセージを表示します。ファイルの作成もモジュールのコピーも行ないません。
−+ +−	置換	TLIBは，指定されたライブラリの指定されたモジュールを入れ替えます。この操作により，削除と追加を一度に行なうことができます。
−* *−	抽出・削除	TLIBは，指定されたライブラリから指定されたモジュールを抜き取ります。この操作により，読み出しと削除を一度に行なうことができます。

削除操作で指定する必要があるのはモジュール名だけです。ドライブ名，パス，拡張子を入力することもできますが，モジュール名以外はすべて無視されます。

ライブラリの中のモジュールの名前を変更することはできません。モジュール名を変更するには，まずそのモジュールを読み出し・削除し，次に作成されたファイルの名前を変更し，最後にそのモジュールをライブラリに追加します。

ライブラリの作成

ライブラリを作成するには，作成しようとする（まだ存在しない）ライブラリに，モジュールを追加します。

レスポンスファイルの使い方

非常に多数の操作をするとき，または1組の操作を繰り返し行なうときは，**レスポンスファイル**を使いはじめたいと思うでしょう。レスポンスファイルは，TLIBコマンドのすべてまたは一部を含むテキストファイルです。レスポンスファイルを使うと，DOSのコマンドライン上で指定できるよりも長いTLIBコマンドを指定できます。

レスポンスファイルを使うには，TLIBのコマンドライン上のどこでも好きなところに@<*pathname*>と指定してください。

- ■レスポンスファイルには，複数のテキスト行を入れることができます。次の行に続くことを示すには，行の最後にアンド記号（&）をつけてください。
- ■TLIBのコマンドラインの全体を，レスポンスファイルの中に入れる必要はありません。レスポンスファイルからTLIBコマンドラインの一部を読み込み，残りの部分をキーボードから入力することもできます。
- ■1つのTLIBコマンドラインに複数のレスポンスファイルを使用することもできます。レスポンスファイルの例は，「TLIBの使用例」の節を参照してください。

文字の大小の区別（/Cオプション）

ライブラリにモジュールを追加すると，TLIBは，そのライブラリのモジュールの中で定義されているすべてのパブリックシンボルの辞書を更新します。ライブラリの中のすべてのシンボルは，互いに異なっていなければなりません。シンボルの重複を引き起こすモジュールを追加しようとしたときは，メッセージを表示し，そのモジュールを追加しません。

通常TLIBは，ライブラリの中のシンボルの重複をチェックするとき，大文字と小文字を区別しません。たとえば，シンボル*lookup*と*LOOKUP*は同じものと見なされます。アセンブラは/**MX**スイッチをつけると大文字と小文字を区別するので，ライブラリのどれかのモジュールで定義されているシンボルと，文字の大小だけが異なるシンボルを含むモジュー

ルを追加しようとするときは，/Cオプションを使う必要があります。/Cオプションを指定すると，TLIBに，ライブラリ中の既存のシンボルと文字の大小だけが異なるシンボルを含むモジュールを追加させることができます。

/Cオプションを指定しないとき，TLIBが文字の大小だけが異なるシンボルを拒絶するのは，一見合理的でない気がします。アセンブラが文字の大小を区別する言語であることを考慮すればなおさらです。その理由は，リンカにはライブラリのシンボルの文字の大小を区別しないものがあるからです。

TLINKには，シンボルの文字の大小の区別をオンにするオプションが用意されていますので，文字の大小だけが異なるシンボルを含むライブラリでも使用できます。そのライブラリをTLINKでしか使用しないなら，/Cオプションを使っても問題はありません。

そのライブラリを他のリンカでも使えるようにしたいときは，/Cオプションを使わないようにしてください。

TLIBの使用例

次にTLIBユーティリティの使い方（コマンドライン）を，例を用いて説明します。

1. MYLIB.LIBという名前のライブラリを作成し，その中にX.OBJ，Y.OBJ，Z.OBJの3つのモジュールを入れたいとき。

```
tlib mylib +x +y +z
```

2. 例1と同じライブラリを作成し，そのリストをファイルに出力したいとき。

```
tlib mylib +x +y +z, mylib.lst
```

3. 既存のライブラリCS.LIBのリストをファイルに出力したいとき。

```
tlib cs, cs.lst
```

4. モジュールX.OBJを新しい版と入れ替え，A.OBJを追加し，Z.OBJを削除したいとき。

```
tlib mylib -+x +a -z
```

5. MYLIB.LIBからY.OBJを読み出し，MYLIB.LIBの内容をファイルに出力したいとき。

```
tlib mylib *y, mylib.lst
```

6. レスポンスファイルを使って，新しいライブラリALPHA.LIBを作成し，その中にA.OBJ, BY.OBJ, ..., G.OBJを入れたいとき。

まず次のようなテキストファイルALPHA.RSPを作成します。

```
+a.obj +b.obj +c.obj &
+d.obj +e.obj +f.obj &
+g.obj
```

次にTLIBコマンド

```
tlib alpha @alpha.rsp, alpha.lst
```

を実行します。

拡張辞書の作成（/Eオプション）

/Eオプションを使うと，ライブラリファイルの拡張辞書を作成し，それをライブラリファイルに連結することができます。この辞書には，標準の辞書には含まれていない情報が，非常に圧縮した形式で入れられます。TLINKは，この情報を使って，ライブラリファイルをより高速で処理することができます。ライブラリファイルがフロッピィディスクや低速のハードディスクに入っているときは，特に効果があります。たとえば，Turbo C 2.0のマスターディスクに含まれているライブラリでは，すべてこの拡張辞書が使用されています。

ライブラリを変更し，変更後の内容について拡張辞書を作成したいときは，モジュールの追加，削除，置換のためにTLIBを起動するときに/Eオプションを指定します。拡張辞書を作成したいが，ライブラリの内容を変更したくないときは，TLIBのコマンドラインに /Eオプションを指定し，さらにそのライブラリに存在しないモジュールを削除するように指定します。TLIBは，指定されたモジュールが指定されたライブラリに見つからなかったことを知らせる警告メッセージを表示しますが，そのライブラリの拡張辞書を作成します。たとえば次のように指定します。

```
tlib /E mylib -bogus
```

GREP

GREPは，一度に複数のファイルの中のテキストを検索することができる強力な検索ユーティリティです。

GREPのコマンドラインの一般的な書式は次のとおりです。

```
GREP [options] searchstring filespec [filespec filespec ... filespec]
```

たとえば，どのソースファイルの中で**setupmodem**関数を呼び出しているか知りたいときは，GREPを使ってすべての.ASMファイルの内容を検索し，文字列**setupmodem**を探すことができます。この例では，次のように入力します。

```
grep setupmodem *.asm
```

GREPのオプション

GREPのオプションは，ダッシュ記号(-)と1つの文字により指定します。オプションの各文字は，ユーザがオン,オフできるスイッチです。文字の後にプラス記号（+）を付けると，そのオプションをオンすることができます。また文字の後にダッシュ（-）を付けると，そのオプションをオフすることができます。

デフォルトはオンです（+も-も指定していないときは，+を指定したものと見なされます)。すなわち**-r**は**-r+**を意味します。複数のオプションを別々に（**-i -d -l**のように）指定することもできますし，組み合わせて（**-idl**または**-il -d**のように）指定することもできます。

GREPのオプション文字とその意味は次のとおりです。

-c　**行数のみ**：マッチする行の数だけが表示されます。GREPは，少なくとも1つのマッチ行を含むファイルの名前とマッチ行の数を表示します。マッチ行の内容は表示されます。

-d　**ディレクトリ**：GREPは，コマンドラインに指定された各*filespec*にマッチするファイルを見つけるために，*filespec*に指定されたディレクトリとその下のすべてのサブディレクトリを探します。*filespec*にパスを指定しなかったときは，GREPは，そのファイルがカレントディレクトリにあるものと見なします。

-i **文字の大小の無視**：GREPは，大文字と小文字を区別しません。どのような場合にも，文字a～zを対応する文字A～Zと同等であるとみなします。

-l **マッチファイルのリスト**：マッチ行を含むファイルの名前だけが表示されます。GREPは，マッチ行が見つかると，そのファイルの名前を表示して，すぐに次のファイルを調べます。

-n **行番号**：表示されるマッチする行の前に行番号が付けられます。

-o **UNIX形式の出力**：UNIX方式のコマンドライン上のパイプ操作を容易にサポートすることができるように，マッチ行の出力形式を変更します。出力されるすべてのマッチする行の前には，その行を含むファイルの名前がつけられます。

-r **正規表現による検索**：*searchstring* は，リテラル文字列ではなく，正規表現として扱われます。

-u **オプションの更新**：GREPは，コマンドラインに指定されたオプションをデフォルトのオプションと組み合わせて，それらを新しいデフォルトとしてGREP.COMファイルに書き込みます。このオプションを使うと，デフォルトのオプションの設定を，自由に変更することができます。

-v **ノンマッチ**：ノンマッチ行だけが表示されます。検索文字列を含まない行が，ノンマッチ行とみなされます。

-w **ワード検索**：正規表現にマッチするテキストは，その直前と直後の文字がワードの一部ではあり得ないときにだけ，マッチと見なされます。デフォルトのワード文字セットは，*A*～*Z*, *0*～*9*および下線（_）です。このオプションの別形により，ワード文字の集合を指定することもできます。書式は**-w**[*set*]です。*set*は，正規表現の集合を指定します。集合の要素に英字を指定したときは，大文字または小文字のどちらを指定しても，また検索が文字の大小を区別する場合にも，大文字,小文字の両方を指定したのと同じ効果があります。**-w**オプションを**-u**オプションと組み合わせると，新しい文字の集合をデフォルトの集合として書き込むことができます。

-z **詳細表示**：GREPは，検索したすべてのファイルのファイル名を表示します。各マッチ行の前には，行番号が表示されます。マッチ行の数も表示されます。マッチする行がない場合には，ゼロが表示されます。

優先順位

コマンドライン上に同じオプションを複数回指定したときは，各指定はそれ以前の指定に優先します。たとえば次のようなコマンドライン

```
grep -r -i- -d -i -r- main( my*.asm
```

では，-dオプションはオンに，-iオプションはオンに，-rオプションはオフになります。

-uオプションを使うと，各オプションのデフォルトの設定をGREP.COMに書き込むことができます。たとえば，いつでも-zオプション（詳細表示）をオンにして検索したいときは，次のコマンドラインを使ってインストールすることができます。

```
grep -u -z
```

検索文字列

検索文字列*searchstring*の値は，GREPが検索するパターンを指定します。-rオプションがオンのときは，検索文字列は正規表現として扱われます。オフのときは，リテラルな文字列として扱われます。正規表現として扱われるときは，いくつかの文字は特別の意味を持ちます。それらの文字は，検索を制御する演算子です。リテラルな文字列として扱われるときは，演算子はありません。各文字は，文字どおりに扱われます。

検索文字列を引用符（" "）で囲むと，空白やタブがデリミタ（区切り文字）として扱われるのを防ぐことができます。マッチングのテストは，行の境界をまたいでは，行なわれません（マッチする文字列は，1つの行に含まれていなければなりません）。

正規表現は，1文字または文字の集合を大カッコ（）で囲んだものです。正規表現を連結したものはまた正規表現になります。

正規表現の演算子

-rオプションをオンにしたときは，検索文字列は正規表現と見なされます。この場合には，次の文字は特別の意味を持ちます。

^　表現の前のキャレット（べき乗記号）は，行の先頭にマッチします。

$　表現の後のドル記号は，行の終わりにマッチします。

.　ピリオドは，任意の1文字にマッチします。

　アスタリスク記号は，直前の表現の任意個数（ゼロを含む）の繰り返しを意味するワイルドカードです。たとえば検索文字列*fo*＊では，アスタリスク＊は表現*o*に作用します。検索文字列は*f, fo, foo,* ..（*f*だけ，または*f*の後に任意の個数の*o*がつづく文字列）にマッチします。ただし*fa*にはマッチしません。

＋　プラス記号は，直前の表現の任意個（1個以上）の繰り返しを意味します。たとえば検索文字列*fo*＋は，*fo, foo,* ..（*f*の後に任意の個数の*o*がつづく文字列）にマッチし

ます。ただしfにはマッチしません。

[] 大カッコで囲んだ文字列は，そのストリングの中のすべての文字にマッチします。文字列の最初の文字がキャレット（ ^ ）のときは，その文字列の中の文字以外のすべての文字にマッチします。たとえば［*xyz*］は*x, y, z*にマッチし，［ ^*xyz*］は*x, y, z*以外のすべての文字にマッチします。2つの文字の間にハイフン (-) を入れることにより，文字の範囲を指定することができます。これらの記号を組み合わせると，複雑な表現を書くことができます。たとえば[*a-bd-z*]は，c以外の小文字の英字と疑問符？にマッチします。

¥ 円記号は，エスケープです。その後に続く文字は，リテラルな文字として扱われます。たとえば¥.は"任意の文字"ではなく，ピリオドにマッチします。

注意：上記の特殊文字のうちの4文字 ($, .,＊ , ＋) は，角カッコの内部にあるときは特別の意味を持ちません。さらにキャレット（^）は，左カッコ（[）の直後にあるときにだけ，特別の意味を持ちます。上の表に掲載されていない普通の文字は，その文字にマッチします（たとえば, ＞は＞にマッチし，#は#にマッチします)。

ファイルの指定

GREPのコマンドラインの3番目の項はファイル指定*filespec*です。*filespec*は，検索すべきファイル（またはファイルのグループ）を指定します。*filespec*は，単一のファイルの名前でもよいし，DOSのワイルドカード（?, ＊）を含む包括的な指定でもかまいません。さらに*filespec*の一部として，パス（ドライブとディレクトリに関する情報）を入力することもできます。*filespec*にパスを指定しないときは，GREPは，カレントディレクトリの中だけを探します。

GREPの使用例

次の例では, GREPのオプションはすべて, デフォルトがオフにセットされているものとしています。

例1

コマンドライン: `grep start: *.asm`

マッチするもの:
```
start:
restart:
```

マッチしないもの:
```
restarted:
ClockStart:
```

検索されるファイル: カレントディレクトリ内の＊.ASM

注意: デフォルトにより，文字の大小が区別される。

例2

コマンドライン: `grep -r [^a-z]main¥ *( *.asm`

マッチするもの:
```
main(i:integer)
main(i, j:integer)
if (main ()) halt;
```

マッチしないもの:
```
mymain()
MAIN(i:integer);
```

検索されるファイル: カレントディレクトリ内の＊.ASM

注意: この例の検索文字列は，次の条件を満足する*main*にマッチします。

・直前の文字が小文字の英字でないこと（[＾a-z]）

・直後に任意個数（ゼロでもよい）の空白が続くこと（¥ ＊）

・その後に左丸カッコが続くこと

空白やタブは，通常，コマンドライン上のデリミタ（区切り文字）と見なされますので，それらを正規表現の一部として含めたいときは，引用形式にしなければなりません。上例では，*main*の後の空白は，円記号（エスケープ文字）により引用形式にされています。次のように空白を二重引用符で囲むことにより，同じ結果を得ることができます。

```
[^a-z]main" "*
```

例3

コマンドライン: `grep -ri [a-c]:¥¥data¥.fil *.asm *.inc`

マッチするもの:
```
A:¥data.fil
C:¥Data.Fil
B:¥DATA.FIL
```

マッチしないもの:

```
d:¥data.fil
a:¥data.fil
```

検索されるファイル: カレントディレクトリ内の＊.ASMおよび＊.INC

注意: 円記号（¥）とピリオド（.）は通常特別の意味を持ちますので，これらの記号を含む文字列を検索したいときは，直前にエスケープ文字（円記号）を入れて，引用形式にしなければなりません。

例4

コマンドライン:

```
grep -ri [^a-z]word[^a-z] *.doc
```

マッチするもの:

```
every new word must be on a new line.
MY WORD!
word--smallest unit of speech.
In the beginning there was the WORD,  and the WORD
```

マッチしないもの:

```
Each file has at least 2000 words.
He misspells toward as torword.
```

検索されるファイル: カレントディレクトリ内の＊.DOC

注意: この例のコマンドラインは，単語を探すための基本的な書式を示します。

例5

コマンドライン:

```
grep -iw word *.doc
```

マッチするもの:

```
every new word must be on a new line. However,
MY WORD!
word: smallest unit of speech which conveys meaning.
In the beginning there was the WORD,  and the WORD
```

マッチしないもの:

```
Each document contains at least 2000 words!
He seems continually misspell "toward" as "torword."
```

検索されるファイル: カレントディレクトリ内の＊.DOC

注意: この例のコマンドラインは,. 単語を探すための基本的な書式を示します。

例6

コマンドライン:

```
grep "search string with spaces" *.doc *.asm a:¥work¥myfile.*
```

マッチするもの:

```
This is a search string with spaces in it.
```

マッチしないもの:

```
THIS IS A SEARCH STRING WITH SPACES IN IT.
This is a search string with many spaces in it.
```

検索されるファイル: カレントディレクトリ内の＊.DOCと＊.ASM，およびドライブA:のディスクのディレクトリ¥WORKの中のMYFILE.＊

注意: このコマンドラインは，空白を含む文字列を探すための基本的な書式を示します。

例7

コマンドライン:

```
grep -rd "[ ,.:?'¥"]"$ ¥*.doc
```

マッチするもの:

```
He said hi to me.
Where are you going?
Happening in anticipation of a unique situation,
Examples include the following:
"Many men smoke,  but fu man chu."
```

マッチしないもの:

```
He said "Hi" to me
Where are you going? I'm headed to the beach this
```

検索されるファイル: ルートディレクトリとルートディレクトリのすべてのサブディレクトリの中の＊.DOC

注意: この例は，行末にある ，．：？’を探します。大カッコ内の2重引用符の前にはエスケープ文字が入力されていますので，この2重引用符は，左角カッコの前の2重引用符と対をなす引用符ではなく，通常の文字として扱われることに注意してください。また引用形式の文字列の外側にドル記号があることに注意してください。これは，正規表現を連結してより長い正規表現を作ることができることを示す一例です。

例8

コマンドライン:

```
grep -ild " the " ¥*.doc         または
grep -i -l -d " the " ¥*.doc     または
grep -il -d " the " ¥*.doc
```

マッチするもの:

```
Anyway,  this is the time we have
do you think? the main reason we are
```

マッチしないもの:

```
He said "Hi" to me just when I
Where are you going? I'll bet you're headed to
```

検索されるファイル: カレントディレクトリおよびカレントディレクトリのすべてのサブディレクトリの中の＊.DOC

注意: この例は，文字の大小を無視して検索し，少なくとも1つのマッチする行を含むファイルの名前だけを表示します。コマンドラインの3つの例は，複数のオプションの異なる指定方法を示します。

OBJXREF

OBJXREFは，オブジェクトファイルやライブラリファイルを調べて，シンボルやセグメントに関する情報をレポートするユーティリティです。OBJXREFの作成するレポートの1つは，パブリック名と**パブリック名**の参照のリストです。もう1つは，**オブジェクトモジュール**により決められるセグメントのサイズのリストです。

パブリック名には2種類あります。グローバル変数と関数名です。後述する「OBJXREFレポートの例」の節では，TEST1.ASMおよびTEST2.ASMの2つのファイルを使って，各ファイルのパブリック名とそれらに対する外部からの参照について説明します。

オブジェクトモジュールは，TASM, TC, TCCなどにより作成されるオブジェクトファイル（.OBJ）です。ライブラリ（.LIB）ファイルは，複数のオブジェクトモジュールを含みます。TASMにより作成されたオブジェクトモジュールは，コマンドライン上に異なるオブジェクトファイル名を指定しないかぎり，ソースファイル（.ASM）と同じ名前が付けられます。

OBJXREFのコマンドライン

OBJXREFのコマンドラインは，コマンド名OBJXREFの後に，コマンドラインオプションと，オブジェクトファイル名およびライブラリファイル名のリストを指定します。各項目は空白またはタブにより分離されます。コマンドラインの書式は次のとおりです。

```
OBJXREF < options > filename <filename ... >
```

コマンドラインオプションは，OBJXREFが作成するレポートの種類と，詳細度を決定します。コマンドラインオプションについては，次の「コマンドラインオプション」のセクションで詳しく説明します。

各オプションは，スラッシュ（/）で始まり，その後に1文字または2文字のオプション名が続きます。

*filename*は，オブジェクトファイルまたはライブラリファイルの名前です。*filename*は，コマンドライン上に指定してもよいし，レスポンスファイルの中に指定することもできます。コマンドライン上では，ファイル名を空白またはタブで分離しなければなりません。.OBJファイルとして指定したすべてのオブジェクトモジュールは，その情報がレポートされます。それに対して.LIBファイルのモジュールは，パブリック名が参照されるものしか

レポートされません。

一般則として，プログラムを正しくリンクするために必要なすべての.OBJと.LIBファイルをリストしなければなりません。

ファイル名には，ドライブ名とディレクトリパスを含めることができます。DOSのワイルドカード文字（?，*）を使って複数のファイルを指定することもできます。ファイル名*filename*は，.OBJオブジェクトファイル名またはライブラリファイル名のどちらかです。拡張子を指定しなかったときは，拡張子は.OBJであるとみなされます。

オプションとファイル名は，コマンドライン上にどのような順番で指定してもかまいません。

OBJXREFのレポートは，DOSの標準出力に書き込まれます。デフォルトは表示画面です。DOSのリダイレクション文字（>）を使うと，プリンタに出力したり（>*PRN*），ファイルに書き込んだり（>*lstfile*）することができます。

オプションもファイル名も指定しないでOBJXREFだけを入力すると，使用可能なオプションの要約説明が表示されます。

コマンドラインオプション

OBJXREFのコマンドラインオプションは，2種類あります。1つは制御オプション，もう1つはレポートオプションです。

制御オプション

制御オプションは，OBJXREFのデフォルトの動作を変更します（どの制御オプションも指定されていないときの動作がデフォルトの動作です）。

/I　パブリック名の文字の大小を無視する：TLINKでリンクするときに/**c**オプション（文字の大小の区別をオンにする）を指定しない場合には，このオプションを使用してください。

/F　ライブラリの全モジュールについてレポートする：指定された.LIBファイルのすべてのモジュールの情報をレポートします。すなわちパブリック名が参照されるモジュールについてだけでなく，参照されないモジュールについてもレポートされます。このオプションを使うと，ライブラリファイルの全内容に関する情報を得ることができます（「OBJXREFの使用例」の節の例4を見てください）。

/V　詳細出力：読み込んだファイルの名前をリストし，パブリック名，モジュール，セグメント，クラス等のすべての情報を表示します。

/Z　長さゼロのセグメント定義をレポートに含める：オブジェクトモジュールは，セグメントを定義していても，その中のスペースをまったく使用していないことがあります。これらの長さゼロのセグメント定義をリストに含めると，通常，モジュールサイズのレポートが使いにくくなります。ただしあるセグメントのすべての定義を取り除きたいときに便利です。

レポートオプション

レポートオプションは，作成されるレポートの種類と，レポートの詳細さのレベルを制御します。

/RC　クラス別レポート：各モジュールのセグメントのサイズが，セグメントクラスごとにリストされます。

/RM　モジュール別レポート：パブリック名が，モジュール別にリストされます。

/RP　パブリック名リポート：パブリック名とそれが定義されているモジュール名が，パブリック名の順にリストされます。

/RR　参照レポート：参照されるパブリック名とそれを外部参照するモジュール名が，パブリック名ごとにリストされます（デフォルトのレポート）。

/RS　モジュールサイズレポート：モジュールサイズが，セグメント別にリストされます。

/RU　参照されないシンボルのレポート：参照されないパブリック名が，定義されているモジュールごとにリストされます。

/RV　詳細レポート：全種類のレポートが作成されます。

/RX　外部参照レポート：各モジュールが外部参照するパブリック名がリストされます。

レスポンスファイル

コマンドラインは，DOSにより最大128文字に制限されています。オプションおよびファイル名のリストがこの上限を越えるときは，ファイル名を**レスポンスファイル**に置かなければなりません。

レスポンスファイルは，テキストエディタで作成することができるテキストファイルです。OBJXREFは，複数の拡張子を持つレスポンスファイルを識別することができます。

レスポンスファイルは，次の節で説明するオプションのどれかを使って，コマンドラインから呼び出すことができます。レスポンスファイルの名前は，オプションの後に空白を

置かずに指定します（たとえば/ Lではなく/L)。

コマンドラインには，複数のレスポンスファイルを指定することができます。レスポンファイルの前後に，さらに別の.OBJファイルや.LIBを指定することもできます。

自由形式のレスポンスファイル

自由形式のレスポンスファイルは，テキストエディタを使って，.EXEファイルを作成するために必要なすべての.OBJファイルと.LIBファイルをリストしたものです。

自由形式のレスポンスファイルを使うには，ファイル名の前に単価記号（@）をつけて，コマンドライン上に指定します。コマンドラインの他の項とは，空白またはタブで分離します。

```
@filename @filename ....
```

注意：レスポンスファイルにリストされているファイルで，拡張子が指定されていないものは，.OBJファイルと見なされます。

リンカのレスポンスファイル

OBJXREFでは，TLINKのレスポンスファイル形式のファイルを利用することもできます。リンカのレスポンスファイルは，/Lオプションにより呼び出します。

```
/Lfilename
```

これらのファイルの使い方については，「OBJXREFの使用例」の例2を参照してください。

/Oオプション

OBJXREFに，カレントディレクトリ以外のディレクトリの中の.OBJファイルを探させたいときは，/Oの後に続けてそのディレクトリ名を指定します。

```
/Omyobjdir
```

/Nオプション

/Nオプションの後に続けてモジュール，セグメント，クラス，またはパブリック名を指

定することにより，OBJXREFは，指定されたモジュール，セグメント，クラス，またはパブリック名に関する情報だけをレポートします。たとえば

```
OBJXREF <filelist> /RM /NTest
```

は, OBJXREFに, *Test*という名前のモジュールの情報だけをリストしたレポートを作成するように指示します。

OBJXREFレポートの例

いま，Turbo Assemblerのディレクトリの中に2つのソースファイルがあるとしましょう。そしてOBJXREFを使って, それらのファイルからアセンブルされるオブジェクトファイルに関するレポートを作成するとします。ソースファイルの名前をTEST1.ASMとTEST2.AMSとし，それぞれ次のような内容であるとします。

```
; TEST1.ASM

        .MODEL  SMALL
        STACK   200h

        EXTRN   GOODBY:BYTE                 ; Goodbyeを参照する
        EXTRN   SAYHELLO:NEAR               ; SayHelloを参照する

        PUBLIC  HELLO                       ; Helloをパブリックにする
        PUBLIC NOTUSED                      ; NotUsedをパブリックにする

.DATA
HELLO   DB   'Hello', 10, 13, '$'          ; Helloを定義する
NOTUSED DW   ?
HIDEN   DW   ?

.CODE
SAYBYE  PROC    NEAR                        ; SayBye
        mov     dx, OFFSET GOODBYE
        mov     ah, 9
        INT     21h
        RET
SAYBYE  ENDP

START   PROC    NEAR                        ; Startを定義する
        mov     ax, @data
        mov     dx, ax
        CALL    SAYHELLO                    ; SayHelloを参照する
        CALL    SAYBYE                      ; SayByeを参照する
    EXIT:
        mov     ax, 04C00h
        INT     21h
START   ENDP
        END     START
```

```
; TEST2.ASM

        .MODEL  SMALL

        EXTRN   HELLO:BYTE                  ; Helloを参照する

        PUBLIC  GOODBYE                     ; Goodbyeをパブリックにする
        PUBLIC  SAYHELLO                    ; SayHelloをパブリックにする

.DATA
GOODBY  DB      'Goodbye', 10, 13, '$'      ; Goodbyeを定義する

.CODE
SAYHELLO        PROC    NEAR                ; SayHelloを定義する
        mov     dx, OFFSET HELLO            ; Helloを参照する
        mov     ah, 9
        INT     21h
        RET
SAYHELLO        ENDP
        END
```

これらのソースファイルからアセンブルされたオブジェクトモジュールを,それぞれTEST1.OBJとTEST2.OBJとします。

OBJXREFのコマンドラインに,この2つのファイル名とレポートタイプ(/Rオプション)を指定することにより,これらの.OBJファイルについて指定した種類のレポートを作成することができます。

注意：以下の例では，OBJXREFの出力の一部分だけを示します。

パブリック名レポート（/RP）

リポートされるオブジェクトモジュールの中で定義されているパリック名と，それらが定義されているオブジェクトモジュール名が，パブリック名についてアルファベット順にリストされます。　コマンドラインに次のように指定すると，

```
OBJXREF /RP test1 test2
```

OBJXREFは，次のようなレポートを作成します。

```
---------------------------------
 Symbol              Defined in
---------------------------------
 GOODBYE               TEST2
 HELLO                 TEST1
 NOTUSED               TEST1
 SAYHELLO              TEST2
---------------------------------
```

モジュール別レポート（/RM）

レポートされるオブジェクトモジュールごとに，その中で定義されているパブリック名がリストされます。

コマンドラインに次のように指定すると，

```
OBJXREF /RM test1 test2
```

OBJXREFは，次のようなレポートを作成します。

```
Module: TEST1 defines the following symbols.

    Hello
    NOTUSED

Module: TEST2 defines the following symbols.

    GOODBYE
    SAYHELLO
```

参照レポート（/RR）（デフォルト）

各パブリック名と，そのパブリック名が定義されているモジュール名が，同じ行に表示されます(モジュール名はカッコに囲んで表示されます)。このパブリック名を外部から参照するモジュールの名前(もしあれば) が，その下の行にインデント（字下げ）して表示されます。

コマンドラインに次のように指定すると，

```
OBJXREF /RR test1 test2
```

OBJXREFは，次のようなレポートを作成します。

```
GOODBYE (TEST2)
        TEST1
HELLO (TEST1)
        TEST2
NOTUSED (TEST1)
SAYHELLO (TEST2)
        TEST1
```

外部参照レポート（/RX）

レポートされるオブジェクトモジュールごとに，その中の外部参照がリストされます。
コマンドラインに次のように指定すると，

```
OBJXREF /RX test1 test2 CS.LIB
```

OBJXREFは次のようなレポートを作成します。

```
Module: TEST1 references the following symbols:

    GOODBYE
    SAYHELLO

Module: TEST2 references the following symbols:

    HELLO
```

モジュールサイズレポート（/RS）

定義されている各セグメント名の下に，そのセグメントを定義しているオブジェクトモジュールの名前とそれらのモジュールで使用されているそのセグメントのサイズがリストされます。サイズはバイト数で，10進数表示または16進数表示の両方で表示されます。そのモジュールのそのセグメント内のシンボルのどれにも初期値が指定されていないときは*"uninitialized"*と表示されます。.ASMファイルの中で絶対アドレスに定義されているセグメントは，セグメントサイズの左側に*Abs*と表示されます。
コマンドラインに次のように指定すると，

```
OBJXREF /RS test1 test2
```

OBJXREFは次のようなレポートを作成します。

```
;Module sizes by segment
STACK
              512 (00200h)   TEST1, uninitialized
              512 (00200h)   total
_DATA
               12 (0000Ch)   TEST1
               10 (0000Ah)   TEST2
               22 (00016h)   total
_TEXT
               24 (00018h)   TEST1
                8 (00008h)   TEST2
               32 (00020h)   Total
```

クラス別レポート（/RC）

セグメントのサイズを，セグメントのクラスごとにリストします。CODEクラスは命令のセグメント，DATAクラスは初期化されるデータのセグメント，BSSクラスは初期化されないデータのセグメントです。クラスタイプが指定されていないセグメントは“No class type”．という見出しの下にリストされます。

コマンドラインに次のように指定すると，

```
OBJXREF /RC test1 test2
```

OBJXREFは次のようなレポートを作成します。

```
;Module sizes by class
CODE
               24 (00018h)    TEST1
                8 (00008h)    TEST2
               32 (00020h)    total
DATA
               12 (0000Ch)    TEST1
               10 (0000Ah)    TEST2
               22 (00016h)    total
STACK
              512 (00200h)    TEST1, uninitialized
              512 (00200h)    total
```

参照されないシンボル名のレポート（/RU）

他のオブジェクトモジュール内で参照されないパブリック名が定義されているモジュールがリストされます。このようなシンボルには，次のものがあります。

■そのシンボル名が定義されているモジュールの内部からだけ参照されており，パブリックシンボルとして定義する必要のないシンボル(そのモジュールのソースがC言語のときは，そのキーワードは**static**として定義しなければなりません。そのモジュールのソースがTASMのときは，そのパブリック定義を削除してください)。

■まったく使用されないシンボル（コードスペースまたはデータスペースを節約するために削除することができます)。

コマンドラインに次のように指定すると，

```
OBJXREF /RU test1 test2
```

OBJXREFは次のようなレポートを作成します。

```
Module:
    TEST1 defines the following unreferenced symbols:
    NOTUSED
```

詳細レポート（/RV）

/RVオプションを指定すると，各タイプのレポートが1つずつ生成されます。

OBJXREFの使用例

次の例では，TEST1.OBJ，TEST2.OBJファイルはカレントディレクトリに，ライブラリファイルはLIBディレクトリに入っているものとしています。

例1　`A>OBJXREF test1 test2 ¥lib¥io.ib`

TEST1.OBJ，TEST2.OBJファイルに加えてさらに，ライブラリファイル¥LIB¥IO.LIBが指定されています。レポートの種類は指定されていませんので，作成されるレポートはデフォルトの参照レポートです。外部参照される各パブリック名について，そのパブリック名を外部参照するモジュールがリストされます。

例2　A>OBJXREF /RV /Ltest1.arf

TEST1.arfはTLINKのレスポンスファイルで，次のように指定されているものとします。

```
test1 test2
test1.exe
test1.map
¥lib¥io
```

この例では /**RV**オプションが指定されていますので，例1と同じオブジェクトモジュールおよびライブラリファイルについて，総合レポートが作成されます。

例3　A>OBJXREF /F /RV ¥lib¥IO.lib

この例では，ライブラリファイルIO.LIBのすべてのモジュールについて総合レポートが作成されます。OBJXREFは，指定されているファイルが完全なプログラムでないときも，レポートを作成することができます。

OBJXREFのメッセージ

OBJXREFは，エラーメッセージと警告メッセージの2種類のメッセージを表示します。

エラーメッセージ

Out of memory

メモリが足りないことを示します。OBJXREFはRAMメモリの中で相互参照を行ないますので，リストされているファイルがTLINKによりメモリ不足を引き起こすことなくリンクできる場合にも，メモリが足りなくなることがあります。このエラーが発生したときは，OBJXREFはレポートの作成を中止します。メモリに常駐しているプログラムを取り除いてスペースを広げるか，RAMを増設してください。

警告メッセージ

WARNING: Unable to open input file rrrr

入力ファイル*rrrr*が見つからないか，またはオープンできませんでした。OBJXREFは，次のファイルの処理に進みます。

WARNING: Unknown option - oooo

オプション名*oooo*の指定が正しくありません。OBJXREFは，このオプションを無視し

ます。

WARNING: Unresolved symbol nnnn in module mmmm

モジュール*mmmm*の中で参照されているパブリック名*nnnn*は，どの.OBJファイルおよび.LIBファイルにも定義されていません。OBJXREFは，どの種類のレポートにおいても，そのシンボルが参照されているが定義されていないことを示す目印をつけます。

WARNING: Invalid file specification ffff

ファイル名*ffff*の指定に誤りがあります。OBJXREFは，次のファイルの処理に進みます。

WARNING: No files matching ffff

コマンドラインまたはレスポンスファイルにリストされたファイル*ffff*が見つからないか，またはオープンできません。OBJXREFは，次のファイルの処理に進みます。

WARNING: Symbol nnnn defined in mmmm1 duplicated in mmmm2

パブリック名*nnnn*が，モジュール*mmmm1*と*mmmm2*の両方で定義されています。OBJXREFは，2番目の定義を無視します。

TCREF

TCREF（ソースモジュールクロスリファレンスユーティリティ）は，2種類のレポートを作成します。1つは，すべてのグローバルシンボルについて，どのモジュールで定義され，どのモジュールで参照されているかを示すクロスリファレンスリストです。もう1つは，各モジュールの中で使用されているシンボルのリストです。

TCREFは，TASMにより作成された一群の.XRFファイルを，入力として受け入れます。これらの.XRFファイルには，各モジュールのクロスリファレンス情報が入っています。TCREFは，これらの入力ファイルから，1種類または2種類のレポートを含む単一のテキストファイルを作成します。TCREFのコマンドラインの書式は次のとおりです。

```
TCREF <XRFファイルのリスト> ',' <REFファイル名> <スイッチ>
```

各XRFファイルはプラス記号(＋)で区切ります。たとえば次のように指定すると，TCREFは，FOO1.XRF，FOO2.XRF，FOO3.XRFを入力として，FOO.REFを出力します。

```
TCREF foo1+foo2+foo3,foo
```

レスポンスファイル

TCREFのコマンドラインは，テキストファイル（レスポンスファイル）を使って指定することもできます。コマンドラインにレスポンスファイル名を指定するときは，ファイル名の前に単価記号（@）を付けます。たとえば，レスポンスファイルDOFOOの内容が

```
foo1+foo2+foo3,foo
```

のとき，コマンドラインに

```
TCREF @dofoo
```

と入力すると，前の例と同じことをします。

TLINKとの互換性

TCREFは，TLINKのコマンドラインの指定をそのまま受け入れます。その場合には，

関係のないスイッチ（リンカの機能だけにしか影響を及ぼさないスイッチなど）やフィールド（ライブラリファイルやMAPファイルなど）の指定は無視します。また，見つからないXRFファイルがあるときは，それを無視します。

TLINKのレスポンスファイルを使うときは，レスポンスファイルの中のファイル名の拡張子をそのまま使用しないでください。なぜなら，それらの拡張子がTCREFのデフォルトの拡張子に優先するため，重大な結果を招く可能性があるからです。たとえばTLINKのレスポンスファイルの内容が

```
foo1+foo2+foo3,foo.exe
```

であるとすると，このレスポンスファイルをそのままTCREFに使わないでください。なぜなら，TCREFは結果をFOO.EXEという名前のファイルに書き込むので，ユーザのプログラムに上書きする可能性があるからです。

スイッチ

TCREFは，すべてのTLINKのスイッチを受け入れますが，そのほとんどを無視します。TCREFは次のスイッチだけを使います

■**/c:** グローバルレポートで文字の大小を区別する。
■**/r:** 指定されてすべてのモジュールにたいしてローカルレポートを作成する。
■**/p#:** レポートの1ページの行数を#行にする。
■**/w#:** レポートの1行の文字数を#文字にする。

出力

名前が同じでも意味が異なるシンボルを混同すると，クロスリファレンス情報が役に立たなくなります。したがってTCREFは，レポートを作成するときに，シンボルのスコープ（通用範囲）を考慮に入れます。クロスリファレンス情報は，いつでも，シンボルを参照するソースファイルとその行番号を知ることができるようリストされます。

グローバル（リンカスコープ）レポート

TCREFのグローバルレポートには，グローバルシンボルの相互参照情報がリストされま

す。文字の大小を区別したレポートを作成したいときは，/cスイッチを指定してください。

このレポートでは，グローバルシンボルは左欄にアルファベット順に現われます。参照は，ソースファイルごとに，右欄にリストされます。番号記号 (#) は，定義がその行にあることを示します。

次にグローバルレポートのプリントアウトの例を示します。

```
Gobal Symbols        Cref # = definition
BAR                  TEST.ASM:  1  3  6  9  12  15  18 +
                                            21  23  29
                                     # TEST2.ASM:  2  4  6  #8
```

このレポートのTEST2.ASMの前の番号記号は，シンボルBARが，そのソースファイルの中で定義されていることを示します。各ソースファイル名の後の数字のリストは，左欄のシンボルが参照されているソース行の番号です。行番号のリストは，TEST.ASMのリストのように，2行以上になることがあります。プラス記号 (+) は，その行が次行に続くことを示します。最後の行の8の前の番号記号 (#) は，BARがTEST2.ASMの行8で定義されていることを示します。

ローカル（モジュールスコープ）レポート

コマンドラインに/rを指定すると，各モジュールのローカルレポートが作成されます。このレポートには，そのモジュール内で使用されているすべてのシンボルが，アルファベット順にリストされます。/cスイッチは，これらのレポートには効果がありません。なぜならそのモジュール内で文字の大小を区別するかしないかは，アセンブリ時に決められているからです。

グローバルレポートと同じように，ローカルレポートでも，右欄はソースファイルごとにリストされます。

次にローカルレポートのプリントアウトの例を示します。

```
Module TEST.ASM Symbols    Cref # = definition
UGH                        TEST.ASM:  1  3  6  9  12  15  18 +
                                                  21  23  29
                                               # UGH.INC:  #2
```

付録E

エラーメッセージ

この章では，Turbo Assemblerのすべてのメッセージを解説します。通常は，メッセージは画面に表示されますが，DOSの標準リダイレクト機能を使ってファイルまたはプリンタへ出力することもできます。その場合は，コマンドライン上で不等号（>）の後にデバイス名またはファイル名を指定します。次にリダイレクトの例を示します。

```
TASM MYFILE > ERRORS
```

Turbo Assemblerのメッセージは，次の4種類にわけることができます。

■指示メッセージ
■警告メッセージ
■エラーメッセージ
■致命的エラーメッセージ

指示メッセージ

指示メッセージには2種類あります。1つはソースファイルのアセンブルときに表示され，もう1つはファイルのアセンブル終了ときに表示されます。次にTurbo Assemblerのスタートアップメッセージを示します。

```
Turbo Assembler Version 1.00 Copyright (C) 1988 Borland International
Assembling file: TEST.ASM
```

Turbo Assemblerがソースファイルのアセンブルを終了すると，アセンブルプロセスをまとめて示すメッセージを表示します。次にそのメッセージを示します。

```
Error message: None
Warning message: None
Remaining memory: 297K
```

/Tコマンドラインオプションを使うと，すべての指示メッセージを消すことができます。この/**T**オプションは，アセンブル中にエラーが発生しなければ指示メッセージだけを消し，エラーが発生した場合は何もしないで通常のスタートアップメッセージと終了メッセージを表示します。

警告メッセージとエラーメッセージ

警告メッセージは，ソースファイルのアセンブル中に何らかの好ましくない状況が発生したことを知らせます。たとえば，Turbo Assemblerがアセンブルときに行なう想定処理（たいていの場合は正しくても常に正しいとは限らない）に不都合が生じた場合にこのメッセージが表示されます。この場合は，生成されたコードが適切なものかどうかをチェックして，警告メッセージの原因をつきとめなければなりません。警告メッセージが表示された場合でも，オブジェクトファイルの生成は継続されます。次に警告メッセージの一般形式を示します。

```
**Warning** filename(line) message
```

マクロ命令や繰り返しブロックの展開中に警告メッセージが発生した場合は，次のようにマクロ名とエラー発生箇ところが示されます。

```
**Warning** filename(line) message macroname(macroline) message
```

これに対して，エラーメッセージが発生した場合は，オブジェクトファイルの生成は中止されます。次にエラーメッセージの一般形式を示します。

```
**Error** filename(line) message
```

マクロ命令や繰り返しブロックの展開中にエラーメッセージが発生した場合は，次のようにマクロ名とエラー発生箇ところが示されます。

```
**Error** filename(line) message macroname(macroline) message
```

以下に警告メッセージとエラーメッセージの一覧（アルファベット順）を示します。

Argument needs type override

命令の構文からサイズを判断できないので，サイズまたは型を指定してください。

例:

```
mov  [bx],1
```

この場合は，次のようにPTR演算子でオペランドのサイズを指定すればエラーを訂正できます。

```
mov  WORD PTR[bx],1
```

Argument to operation instruction has illegal size

サポートされていない処理や命令が指定されています。

例:

```
Q LABEL QWORD
QNOT = NOT Q          ; QWORDを否定することはできない
```

Arithmetic overflow

算術オーバフローが発生しました。

例:

```
X = 20000h * 20000h     ; 32ビットをオーバーフローする
```

ASSUME must be segment register

ASSUME疑似命令に，セグメントレジスタ以外のレジスタが指定されています。

例:

```
ASSUME ax:CODE
```

ASSUME疑似命令には，セグメントレジスタだけを使うことができます。

Assuming segment is 32 bit

80386命令を有効にした後，**SEGMENT**疑似命令でセグメントが定義されましたが，**USE16**または**USE32**疑似命令によってセグメントの型（16または32ビット）が指定されていません。

このような場合は，Turbo Assemblerは32ビットセグメントを想定します（80386プロセッサが32ビットセグメントの命令を実行することを確認しない限り）この型のコードセグメントはDOSのもとで正常に動作する保証はありません。そのための警告がこのメッセージです。

このメッセージを防ぐには，**SEGMENT**疑似命令の引数として**USE16**を指定します。

Bad keyword in SEGMENT statement

SEGMENT疑似命令のアラインメント型または結合型を指定する引数が無効です。

例:

```
DATA SEGMENT PAFA PUBLIC     ; PAFAはPARAの間違い
```

Can't add relative quantities

意味のない2つのオフセットどうしを加算しています。

例:

```
ABC  DB  ?
DEF = ABC + ABC          ; エラー，オフセットは加算できない
```

2つのオフセットどうしの減算，定数とオフセットの加算は可能です。

例:

```
XYZ     DB  5 DUP (0)
XYZEND EQU $
XYZLEN = SYZEND - XYZ     ; 2つのオフセットの減算は有効
XYZ2 = XYZ + 2            ; オフセットと定数の加算も有効
```

Can't address with currently ASSUMEd segment registers

変数参照命令に，セグメントレジスタが指定されていません。

例:

```
DSEG SEGMENT
  ASSUME DS:DSEG
  mov  SI,MPTR      ; XSEGセグメントにアクセスするための
                    ; セグメントレジスタの指定がない
DSEG ENDS
XSEG SEGMENT
MPTR DW  ?
XSEG ENDS
```

Can't convert to pointer

命令オペランドに，メモリポインタに変換できない変数が指定されています。

例: **PTR**演算子を使う場合

```
mov cl,[BYTE PTR al]    ; ALをポインタに変換できない
```

Can't emulate 8087 instruction

Turbo Assemblerでは，/**E**コマンドラインオプションまたは**EMUL**疑似命令を指定することによって，エミュレータ用の浮動小数点命令を生成することができます。しかしカレント命令は，エミュレートすることはできません。

例:

```
EMUL
FNSAVE [WPTR]          ; この命令はエミュレートできない
```

浮動小数点エミュレータは，以下の命令をサポートしていません。

FASAVE, FNSTCW, FNSTENV, FNSTSW

Can't make variable public

変数がパブリックにできないように宣言されています。

例:

```
EXTRN   ABC:NEAR
PUBLIC ABC                 ; エラー，すでにEXTRNとして宣言されている
```

Can't override ES segment

カレント命令に，無効または不正なオーバライドが指定されています。

例:

```
STOS   DS:BYTE PTR[di]
```

STOS命令でデスティネーションアドレスにアクセスする場合は，ESレジスタだけを使うことができます。

Can't subtract dissimilar relative quantities

減算に指定できない2つの値が減算に使われています。

例: 別のセグメント内にある変数どうしの減算

```
SEG1  SEGMENT
A:
SEG1  ENDS
SEG2  SEGMENT
B:
    mov   ax,B-A      ; エラー，AとBは別のセグメントにある
SEG2  ENDS
```

Can't use macro mane in expression

命令オペランドに，マクロ名が使われています。

例:

```
MYMAC      MACRO
    ENDM
    mov ax,MYMAC       ; エラー
```

Can't this outside macro

マクロ定義の中でしか使えない疑似命令がマクロ定義の外で使われています。

例: ENDMやEXITMなどの疑似命令

```
DATA SEGMENT
     ENDM                 ; エラー，マクロ定義の中でのみ使える
```

Code or deta emission to undeclared segment

SEGMENT疑似命令で宣言されたセグメントの外側に，コードまたはデータを生成する命令があります。

例:

```
; ファイルの先頭行
    INC   bx           ; エラー，セグメントの外
    END
```

コードまたはデータは，セグメント内でのみ生成できます。

Constant too large

定数のフォーマットは適切ですが，その値が大きすぎます。

：たとえば**.386**または**.386P**疑似命令によって80386の命令が有効になっている場合にのみ，0ffffhより大きい定数を指定することができます。

CS not correctly assumed

near **CALL**またはnear **JMP**で別のセグメントへジャンプすることはできません。

例:

```
SEG1   SEGMENT
LAB1   LABEL NEAR
SEG1   ENDS
SEG2   SEGMENT
    JMP LAB1          ; エラー，不正セグメント
SEG2   ENDS
```

このエラーはMASMモードでのみ発生します。Idealモードでは，このエラーは適切に処理されます。

CS override in protected mode

P286または**P386**疑似命令によって，286または386の保護モード用の命令のアセンブルが有効になっている場合は，CSオーバライドを必要とする命令は使えません。

例:

```
    P286
    .CODE
CVAL DW   ?
    mov   CVAL,1         ; CSオーバライドが必要
```

/Pコマンドラインオプションを指定すると，この警告メッセージの表示が有効になります。保護モードでは，CSオーバライドを必要とする命令は特別な処置を施さない限り使うことができません。

CS unreachable from current segment

コロン（:）とLABELまたはPROCを使ってコードラベルを定義した場合は，CSレジスタがカレントコードセグメントまたはそれを含むグループをポイントするとは限りません。

例:

```
PROG1    SEGMENT
    ASSUME CS:PROG2
START:                  ; エラー，ASSUMEに不正なCS指定がある
```

このエラーはMASMモードでのみ発生します。Idealモードでは，このエラーは適切に処理されます。

Declaration needs name

疑似命令に，必要なシンボル名が指定されていません。

例:

```
PROC                    ; エラー，PROCにはシンボル名が必要
    RET
ENDP
```

SEGMENT, PROC, STRUC疑似命令には，シンボル名を指定する必要があります。MASMモードではシンボル名は疑似命令より先に指定しますが，Idealモードではシンボル名は疑似命令の後に指定します。

Directive ignored in Turbo Pascal model

Turbo Pascalとリンクするアセンブラモジュールに対して使用できない疑似命令が使われています。**.MODEL**疑似命令でTurbo Pascalを指定する方法については第3章を，Turbo Pascalとのリンク方法についての詳細は**ユーザーズガイド**の第7章をそれぞれ参照してください。

Directive not allowed inside structure definition

STRUC定義ブロック内で無効または不正な疑似命令が指定されています。

例:

```
X STRUC
MEM1 DB   ?
   ORG $+4        ; エラー, STRUC内でORGは使えない
MEM2 DW   ?
ENDS
```

構造体をネストする場合，構造体の宣言の中で別の構造体を宣言することはできません。

例:

```
FOO STRUC
    FOO2 STRUC          ; ネストの内側で構造体の宣言はできない
    ENDS
ENDS
```

構造体の中で別の構造体名を使いたい場合は，はじめにすべての構造体を宣言してください。そして構造体の中で別の構造体の名前を記述してください。

Duplicate dummy argument: *ident*

MACRO疑似命令で定義されたマクロに，複数の同一名の仮引数が指定されています。

例:

```
XYZ  MACRO A,A       ; エラー, 仮引数の二重定義
    DB A
    ENDM
```

マクロ定義内の複数の仮引数に同じ名前を付けてはなりません。

ELSE or ENDIF without IF

ELSEまたはENDIF疑似命令と対（ペア）になる，条件アセンブルブロックを開始するIF疑似命令が見つかりません。

例:

```
BUF DB 10 DUP (?)
    ENDIF               ; エラー, 対応するIFxxxがない
```

Expecting offset quantity

カレント命令に，セグメント内オフセットを指定する正しい形式のオペランド記述がありません。

例:

```
CODE SEGMENT
    mov   ax,LOW CODE
CODE ENDS
```

Expecting offset or pointer quantity

カレント命令に，特定のセグメント内オフセットを指定する正しい形式のオペランド記述がありません。

例:

```
CODE SEGMENT
    mov   ax,SEG CODE      ; エラー，CODEはセグメント名であって，
                           ;         セグメント内のアドレスではない
CODE ENDS
```

Expecting pointer type

カレント命令に，メモリを参照するオペランド記述がありません。

例:

```
LES   di,4            ; 4は定数のため，エラー
```

Expecting scalar type

命令のオペランドまたは演算子に，定数の指定がありません。

例:

```
BB   DB   4
    ROL   ax,BB        ; ROL命令には定数が必要
```

Expecting segment or group quantity

カレント命令に，必要なセグメント名またはグループ名の指定がありません。

例:

```
DATA SEGMENT
    ASSUME   DS:FOO      ; エラー，FOOはグループ名でもセグメント名でもない
FOO   DW   0
DATA ENDS
```

Extra characters on line

命令または式中に，余分な文字があります。

例:

```
ABC = 4   SHL  3  3         ; 3と3の間の演算子が欠落している
```

このエラーは，式パーサの処理続行を不能にする別のエラーに関連して発生します。

Forward reference needs override

前方参照の変数を含む命令を使うと，より多くのコードが必要となります。このメッセージは，JMPやCALL命令にfarアドレスを指定した場合やセグメントオーバライドが必要な場合に表示されます。

例:

```
    ASSUME cs:DATA
    CALL A                  ; nearコールを予想
A PROC FAR                  ; farコール
    mov   ax,MEMVAR         ; オーバライドが必要かどうか不明
DATA   SEGMENT
MEMVAR DW   ?               ; オーバライドが必要であることが判明
```

このような場合は，セグメントオーバライドまたはFARオーバライドを指定してください。

ID not member of structure

Idealモードで，構造体メンバ演算子のピリオド（.）の後に構造体のメンバ名以外のシンボルが指定されています。

例:

```
    IDEAL
STRUC DEMO
    DB   ?
ENDS
COUNT   DW   0
    mov ax,[(DEMO bx).COUNT]     ; COUNTは構造体のメンバ名ではない
```

ピリオドの後には，ピリオドの前に指定された構造体のメンバ名を指定してください

このエラーは，式パーサの処理続行を不能にする別のエラーに関連して発生します。

Illegal forward reference

未定義シンボルが参照されています。疑似命令または演算子の引数は前方参照であってはなりません。

例:

```
IF MYSYM     ; エラー，MYSYMは未定義
        ;
ENDIF
MYSYM EQU 1
```

前方参照は，**IFxxx**疑似命令の引数内で使うことはできません。また**DUP**演算子のカウントとして使うこともできません。

Illegal immediate

命令のオペランドに，無効または不正な即値が使われています。

例:

```
mov  4,al
```

Illegal indexing mode

命令のオペランドに，無効または不正なレジスタの組み合わせが使われています。

例:

```
mov  al,[si+ax]
```

80386以外のプロセッサで有効なインデックスレジスタの組み合わせは，次のとおりです。BX, BP, SI, DI, BX SI, BX+DI, BP+SI, BP+DI

Illegal instruction

ソース行の先頭に，疑似命令でも命令ニーモニックでもない無効または不正なシンボルがあります。

例:

```
move ax,4       ; "mov"が正しい
```

Illegal instruction for currently selected processor(s)

カレントプロセッサで使用できない命令が使われています。

例:

```
.8086
PUSH  1234h     ; 8086では即値はプッシュできない
```

Turbo Assembler起動時は，特に指定のない限り8086プロセッサ命令のアセンブルモードになっています。

186/286/386プロセッサで利用できる拡張命令ニーモニックを使いたい場合は，対応する疑似命令（**P186, P286, P386**）を指定してください。

Illegal local argument

マクロ定義内のLOCAL疑似命令の引数に,無効または不正なシンボル名が指定されています。

例:

```
X    MACRO
     LOCAL 123       ; シンボル名でない
     ENDM
```

Illegal local symbol prefix

LOCAL疑似命令の引数に,無効または不正な文字で始まるローカルシンボルが使われています。

例:

```
LOCALS  XYZ      ; エラー, 2文字でローカルシンボルプレフィクスを示すこと
```

ローカルシンボルの先頭には, __, @@などの有効なシンボル名を付けなくてはなりません（デフォルトは@@です)。

Illegal macro argument

MACRO疑似命令で定義されたマクロ命令に,無効または不正なシンボル名の仮引数が指定されています。

例:

```
X  MACRO   123       ; 不正な仮引数
   ENDM
```

Illegal macro reference

命令のオペランド内で無効または不正なメモリアドレスが使われています。

例:

```
mov   [bx],BYTE PTR A     ; エラー, メモリ間での転送はできない
```

この例では両オペランドがメモリを指しており,MOV命令の正しい形式になっていません。80x86プロセッサファミリでは,命令オペランドの一方でしかメモリアドレスを指定することはできません。

Illegal number

数値内に，無効または不正な文字が使われています。

例:

```
Z = 0ABCGH
```

この例では，16進数に無効な文字“G”が使われています。

Illegal origin address

カレントセグメントのロケーションカウンタ ($) の指定に，無効または不正なアドレスが指定されています。ロケーションカウンタ ($) に対しては，定数，式，カレントセグメント内のシンボル名を指定することができます。

Illegal override in structure

DUP演算子を使って定義された構造体メンバの初期値が指定されています。DUP演算子で定義された構造体メンバを初期化することはできません。

Illegal override register

コロン (:) の前で，セグメントレジスタ (CS, DS, ES, SS, 80386ではさらにFS, GS) 以外のレジスタがセグメントオーバライドとして使われています。

例:

```
mov dx:XYZ,1          ; DXはセグメントレジスタではない
```

Illegal radix

デフォルトの基数を指定する.RADIX疑似命令に，無効または不正な数値が指定されています。

例:

```
.RADIX  7          ; 無効な数値
```

基数は2, 8,10, 16のどれかです。現在の基数の設定にかかわらず，この疑似命令自身に指定された数値は10進数と見なされます。

Illegal register multiplier

レジスタと定数の乗算は，実行することができません。

例:

```
mov ax*3,1
```

レジスタと定数式の乗算が可能な場合は，80386プロセッサで行なうスケールインデックスオペランドの指定時だけです。

Illegal use of constant

命令または式中に，無効または不正な定数が指定されています。

例:

```
mov bx+4,5
```

Illegal use of register

命令または式中に，無効または不正なレジスタ名が指定されています。

例:

```
X = 4 SHL ax        ; SHL演算子にはレジスタを指定できない
```

Illegal use of segment register

カレント命令に，無効または不正なセグメントレジスタ名が指定されています。

例:

```
ADD  SS,4          ; ADDにセグメントレジスタを指定できない
```

Illegal USES register

プロシージャの入口または出口で，無効または不正なレジスタがプッシュまたはポップされています。有効なレジスタは次のとおりです。

AX, BX, CX, DX, SI, DI, DS, ES

.386または**.386P**疑似命令で80386プロセッサを有効にしている場合は，これらのレジスタの32ビットバージョンを使うことができます。

Illegal warning ID

警告メッセージに，無効または不正な3文字識別子が使われています。これらの識別子の一覧は，**ユーザーズガイド**の第3章を参照してください。

Instruction can be compacted with override

前方参照が使われたので，生成コードに**NOP**命令が挿入されます。前方参照を取り除くか，型を指定してください。

例:

```
    JMP X          ; 警告メッセージが出る
    JMP SHORT X    ; 警告メッセージは出ない
X:
```

Invalid model type

モデル疑似命令に，無効または不正なメモリモデルが指定されています。

例:

```
.MODEL GIGANTIC
```

有効なメモリモデルは，タイニィ，スモール，ミディアム，ラージ，ヒュージです。

Invalid operand(s) to instruction

カレント命令に，無効または不正なオペランドの組み合わせが指定されています。

例:

```
FADD ST(2),ST(3)
```

FADD命令では，オペランドの一方にスタックレジスタ名を，他方にスタックの先頭を指定することができます。

Labels can't start with numeric characters

無効または不正な数字または文字を使ったシンボル名があります。

例:　123XYZ

Line too long - truncating

カレントソース行の長さが最大値（255文字）を越えています。余分な文字は無視されます。

Location counter overflow

カレントセグメントがいっぱいになりました。続きのコードまたはデータはセグメントの先頭に上書きされます。

例:

```
ORG 0FFF0h
ARRAY  DW 20 DUP (0)    ; オーバーフロー
```

Missing argument list

繰り返しブロック疑似命令のIRPまたはIRPCに，仮引数と置き換わるための引数が指定されていません。

例:

```
IRP X          ; 引数がない
    DB X
ENDM
```

IRPとIRPC疑似命令を使う場合は，常に仮引数と引数リストを指定してください。

Missing argument size variable

ARGまたはLOCAL疑似命令の＝オプションに続いて，シンボル名が指定されていません。

例:

```
ARG   A:WORD,B:DWORD=      ; エラー，=の後に名前がない
LOCAL X:TBYTE=             ; 同じエラー
```

ARGおよびLOCAL疑似命令でオプションの等号 (=) を指定した場合は，続いてシンボル名を指定してサイズ変数を定義してください。

Missing COMM ID

COMM疑似命令の型指定の前にシンボル名が指定されていません。

例:

```
COMM   NEAR     ; エラー，"NEAR"の前にシンボル名がない
```

COMM疑似命令を使う場合は，型指定の前にシンボル名とコロン（:）を指定してください。

Missing dummy argument

繰り返しブロック疑似命令のIRPまたはIRPCに，仮引数が指定されていません。

例:

```
RP              ; 仮引数がない
    DB X
ENDM
```

IRPとIRPC疑似命令を使う場合は，常に仮引数と引数リストを指定してください。

Missing and quote

文字列または文字定数が引用符で囲まれていません。

例:

```
DB   "abc       ; abcの最後に"がない
mov  al,'X      ; Xの最後に'がない
```

文字列または文字定数は，一対の引用符（' 'または" "）で囲んでください。

Missing macro ID

MACRO疑似命令に，マクロ名が指定されていません。

例:

```
MACRO           ; エラー, 名前がない
    DB A
    ENDM
```

マクロを定義する場合は，マクロの名前を指定してください。

Missing module name

NAME疑似命令の後にモジュール名が指定されていません。**NAME**疑似命令はIdealモードでのみ使うことができます。

Missing or illegal language ID

.MODEL疑似命令の最後に正しい言語識別子が指定されていません。**.MODEL**疑似命令の詳細については，このマニュアルの第3章を参照してください。

Missing or illegal type specifier

カレント命令に，必要な型指定子（**BYTE, WORD**など）が指定されていません。

例:

```
RED LABEL XXX          ; エラー, "XXX"は型指定子ではない
```

Missing term in list

Idealモードで，複数の引数を指定できる疑似命令（**EXTRN, PUBLIC**など）でカンマの後に引数が指定されていません。

例:

```
EXTRN XXX:BYTE,,YYY:WORD
```

Idealモードでは，すべての引数はカンマで区切ってください。ただし引数リストの最後には，カンマを付ける必要はありません。

Missing text macro

疑似命令に，必要な文字列の引数が指定されていません。

例:

```
NEWSTR  SUBSTR        ; エラー，SUBSTRに引数が指定されていない
```

Model must be specified first

メモリモデルを指定せずに簡略セグメント疑似命令が使われています。

例:

```
.CODE      ; エラー，最初に.MODELが指定されていない
```

簡略セグメント疑似命令を使う場合は，最初に**.MODEL**疑似命令でメモリモデルを指定してください。

Name must come first

疑似命令より先に記述すべきシンボル名が，疑似命令より後に記述されています。

例:

```
STRUC ABC          ; エラー，ABCはSTRUCの前に記述しなければならない
```

Idealモードでは，シンボル名は疑似命令の後に指定する規定があります。したがってIdealモードのプログラムをMASMモードでアセンブルすると，必らずこのエラーが発生します。

Need address or register

命令のオペランドフィールドにカンマがあるにもかかわらず，第2オペランドが指定されていません。

例:

```
mov ax,          ; 第2オペランドがない
```

Need angle brackets for structure fill

構造体の定義に，初期値リストが指定されていません。

例:

```
STR1 STRUC
M1   DW   ?
M2   DD   ?
     ENDS
     STR1       ; 初期値リストがない
```

Need colon

EXTRN, GLOBAL, ARG, LOCAL疑似命令の型指定子（BYTE, WORDなど）の後にコロンがありません。

例:

```
EXTRN X BYTE,Y:WORD        ; Xの後にコロンがない
```

Need expression

式中の演算子に，オペランドが指定されていません。

例:

```
X = 4 + * 6
```

Need file name after INCLUDE

INCLUDE疑似命令の後にファイル名が指定されていません。

例:

```
INCLUDE          ; どのファイルをインクルードする?
```

Idealモードでは，ファイル名は引用符で囲んでください。

Need left parenthesis

命令または式中に，必要な左カッコがありません。

例:

```
DB 4 DUP 7
```

DUP演算子の後の式はカッコで囲んでください。

Need pointer expression

このエラーは，Idealモードでのみ発生します。大カッコ（[]）内の式が評価された結果がメモリポインタとなりません。

例:

```
mov ax,[WORD PTR]
```

Idealモードでは，メモリの参照式は常にで囲んでください。

Need quoted string

文字列が引用符で囲まれて正しく指定されていません。Idealモードでは，疑似命令によっては引数を引用符で囲まなくてはなりません。

例:

```
IDEAL
DISPLAY "ALL DONE"
```

Need register in expression

命令または式中に，必要なレジスタが指定されていません。

Need right angle bracket

構造体，共用体，レコードの定義における初期値リストが一対の不等号カッコ（<>）で囲まれていません。

例:

```
MYSTRUC STRUCNAME <1,2,3
```

Need right parenthesis

式中の左カッコと対になる右カッコがありません。

例:

```
X = 5 * (4 + 3
```

カッコは必らず対（ペア）で使ってください。

Need right square bracket

メモリアドレスを参照する式が，一対の大カッコ（[]）で囲まれていません。

例:

```
mov  ax,[SI          ; エラー, SIの後に右大カッコ']'がない
```

大カッコは必らず対（ペア）で使ってください。

Need stack argument

浮動小数点命令のオペランドフィールドにカンマがあるにもかかわらず，第2オペランドが指定されていません。

例:

```
FADD ST,
```

Need structure member name

Idealモードで，構造体メンバ演算子のピリオド(.)の後に構造体メンバ名以外のものが指定されています。

例:

```
    IDEAL
STRUC DEMO
    DB  ?
ENDS
COUNT    DW 0
    mov  ax,[(DEMO bx).]
```

構造体メンバ演算子のピリオド(.)の後には，必らず構造体のメンバ名を指定してください。

Not expectig group or segment quantity

無効または不正なグループ名またはセグメント名が指定されています。

例:

```
CODE SEGMENT
    ROL  ax,CODE    ; エラー，ここでセグメント名は指定できない
```

One non-null field allowed per union expansion

UNION疑似命令を使った共用体の定義において，複数の初期値が指定されています。

例:

```
U     UNION
    DW  ?
    DD  ?
ENDS
UINST U <1,2>     ; エラー，<?,2>または<1,?>が正しい
```

共用体の初期値は，1つだけ設定することができます。

Open conditional

IFxxx疑似命令で開始した条件アセンブルブロックがENDIF疑似命令で終了しないうちに，END疑似命令（ファイルの終わり）に達してしまいました。

例:

```
IF BIGBUG
    END              ; ENDの前にENDIFが必要
```

ENDIF疑似命令で条件ブロックを終了させないで，END疑似命令を指定したことが原因です。

Open procedure

PROC疑似命令で開始したプロシージャがENDP疑似命令で終了しないうちに，END疑似命令（ファイルの終わり）に達してしまいました。

例:

```
MYFUNC PROC
     END             ; ENDの前にENDPが必要
```

ENDP疑似命令でプロシージャを終了させないで，END疑似命令を指定したことが原因です。

Open segment

SEGMENT疑似命令で開始したセグメントがENDS疑似命令で終了しないうちに，END疑似命令（ファイルの終わり）に達してしまいました。

例

```
DATA SEGMENT
     END             ; ENDの前にENDSが必要
```

ENDS疑似命令でセグメントを終了させないで，END疑似命令を指定したことが原因です。

Open structure definition

STRUC疑似命令で開始した構造体がENDS疑似命令で終了しないうちに，END疑似命令（ファイルの終わり）に達してしまいました。

例:

```
X      STRUC
VAL1 DW   ?
     END             ; ENDの前にENDSが必要
```

ENDS疑似命令で構造体を終了させないで，END疑似命令を指定したことが原因です。

Open types do not match

命令のオペランドサイズがもう一方のオペランドサイズと一致していません。または命令の正しいオペランドサイズになっていません。

例:

```
ABC  DB  5
    .
    .
    .
    mov   ax,ABC
```

Pass-dependent construction encountered

Turbo Assemblerの1パス特性が原因で，命令が期待どうおに機能していません。

例:

```
IF1
              ; アセンブリパス(1パス目)でのみ有効
ENDIF
IF2
              ; リスティングパス(2パス目)でのみ有効
ENDIF
```

たいていの場合は，前方参照を取り除くことによってこのエラーを防ぐことができます。

Pointer expression needs brackets

Idealモードにおいて，オペランドにメモリ参照シンボルが指定されていますが，その旨を示すための大カッコが付けられていません。

例:

```
B   DB   0
     mov   al,B     ; Idealモードではエラー，[B]が正しい
```

MASMモードでは大カッコは必要ないので，エラーとはなりません。

Positive count expected

DUP演算子に，0未満のカウント値が指定されています。

例:

```
BUF  -1 DUP(?)     ; エラー，カウント<0 となっている
```

DUPの前のカウント値には，1以上の値を指定してください。

Record field too large

レコード定義において，全フィールド幅の合計が32ビットを超えています。

例:

```
AREC RECORD  RANGE:12,TOP:12,BOTTOM:12
```

Recursive definition not allowed for EQU

EQU定義内でシンボル名が再帰的に使われています。

例:

```
ABC EQU TWOTIMES ABC
```

Register must be AL or AX

ALレジスタまたはAXレジスタしか指定できないオペランドに，それ以外のレジスタが指定されています。

例:

```
IN   CL,dx          ; エラー，IN命令の第1オペランドには
                    ;          アキュムレータのALまたはAXしか使えない
```

Register must be DX

DXレジスタしか指定できないオペランドに，それ以外のレジスタが指定されています。

例:

```
IN   AL,cx          ; エラー，第2オペランドにはCXではなくDXを使うこと
```

Relative jump out of rangeby type nn bytes

条件ジャンプ命令に，カレントロケーションから後方（下位アドレス方向へ）128バイト，前方(上位アドレス方向へ)127バイトを超えるアドレスが指定されています。**USE32**セグメント（32ビットセグメント）内では，条件ジャンプ命令に対して，カレントロケーションから後方（下位アドレス方向へ）32,768バイト，前方（上位アドレス方向へ）32,767バイトを超えるアドレスを指定することができます。

Relative quantity illegal

オペランドに，アセンブル時に不定なメモリアドレスが指定されています。このような指定方法はエラーとなります。

例:

```
DATA SEGMENT PUBLIC
X    DB    0
  IF   OFFSET X GT 127     ; アセンブル時には不定
```

Reserved word used as symbol

Turbo Assemblerの予約語（定義済みシンボル）がユーザシンボル名として使われています。プログラムは正常にアセンブルされますが，予約語をユーザシンボルとして使用することは避けるべきです。

Rotate count must be constant or CL

シフトまたはローテート命令のオペランドに，定数またはCLレジスタ以外のものが指定されています。

例:

```
ROL   ax,DL          ; エラー, DLはカウンタとして使えない
```

シフトまたはローテート命令の第2オペランドには，定数またはCLレジスタだけが指定できます。

Rolate count out of range

シフトまたはローテート命令の第2オペランドの値が大きすぎます。

例:

```
.8086
SHL   DL,3     ; エラー, 8086は1ビットずつシフト可能
.286
ROR   ax,40    ; エラー, シフトの最大値は31
```

8086プロセッサでは，1ビットずつしかシフトできませんが，上位のプロセッサでは最大31のシフトカウントが指定できます。

Segment alignment not strict enough

指定されたアラインメントの値が無効または不正です。2の累乗以外の値が指定されたか，SEGMENT疑似命令で指定されたアラインメント型よりも単位の大きな値が指定されています。

例:

```
DATA SEGMENT PARA
    ALIGN 32          ; エラー, PARAで16バイトごとのアラインメントが
                      ;           すでに指定されている
    ALIGN 3           ; エラー, 2の累乗以外は指定できない
```

Segment attributes illegally redefined

以前定義されたセグメントがSEGMENT疑似命令で再定義され，別の属性が指定されています。

例:

```
DATA SEGMENT BYTE PUBLIC
DATA ENDS
DATA SEGMENT PARA        ; エラー, すでにバイトアラインメントで定義されている
DATA ENDS
```

定義済みセグメントを再定義する場合は，属性を一致させるか省略してください。省略した場合は，最初のセグメント定義で使われた属性が指定されます。

String too long

指定された文字列の長さが最大値（255文字）を超えています。

Symbol already defined: *ident*

定義されたシンボルは，同じ型ですでに宣言されています。

例:

```
BB  DB  1,2,3
BB  DB  ?         ; エラー，BBは定義済み
```

Symbol already different kind

定義されたシンボルは，別の型ですでに宣言されています。

例:

```
BB  DB  1,2,3
BB  DW  ?         ; エラー，BBは定義済み
```

Symbol has no width or mask

WIDTHまたは**MASK**演算子のオペランドが，レコードまたはレコードフィールドの名前ではありません。

例:

```
B  DB  0
    mov ax,MASK B       ; Bはレコードフィールドでない
```

Symbol is not a segment or already part a group

指定したシンボル名は，すでに別のグループの構成セグメントとして定義されています。または指定したシンボル名は，セグメント名ではありません。

例:

```
DATA     SEGMENT
DATA     ENDS
DGROUP     GROUP  DATA
DGROUP2    GROUP  DATA     ; エラー，DATAはすでにDGROUPに属している
```

Too few operands to instruction

命令オペランドの数が足りません。

例:

```
ADD  ax     ; 第2オペランドがない
```

Too many errors or warnings

これ以上エラーメッセージを表示できません。エラーメッセージの数が最大数100を超えています。以後，エラーメッセージではなく，警告メッセージを表示しながらアセンブルを続行します。

Too many initial values

構造体または共用体に対して，そのメンバの数を超える初期値が指定されています。

例:

```
XYZ  STRUC
A1 DB ?
A2 DD ?
XYZ  ENDS
ANXYZ XYZ <1,2,3>    ; エラー, XYZには2つのメンバしかない
```

構造体または共用体には，そのメンバの数を超える初期値を指定することはできません。

Too many register multipliers in expression

80386のスケールインデックスオペランドの複数のレジスタに,スケールファクタが指定されています。

例:

```
mov  EAX,[2*EAX+4*EDX]    ; スケールファクタ数の超過
```

Too many registers in expression

カレント命令に,複数のインデックスレジスタまたはベースレジスタが使われています。

例:

```
mov ax,[BP+SI+DI]    ; SIとDIを同時に使うことはできない
```

Undefined symbol

ソースファイル中に未定義のシンボルがあります。

Unexpected end of file （no END directive）

ソースファイルの最後にEND疑似命令がありません。

ソースファイルの最後には，必らずEND疑似命令を指定してください。

Unknown character

命令または式中に，Turbo Assemblerのシンボルまたは式の命名規定に従わない文字があります。

例:

```
ADD  ax,!1      ; エラー，!はシンボルとして使えない
```

Unmatched ENDP: *ident*

プロシージャを開始した**PROC**疑似命令で指定された名前と**ENDP**疑似命令で指定された名前が一致していません。

例:

```
ABC PROC
XYZ ENDP     ; エラー，XYZはABCの間違い
```

Unmatched ENDS: *ident*

セグメントを開始した**SEGMENT**疑似命令，構造体を開始した**STRUC**疑似命令，共用体を開始した**UNION**疑似命令のどれかで指定された名前と**ENDS**疑似命令で指定された名前が一致していません。

例:

```
ABC STRUC
XYZ ENDS        ; エラー，XYZはABCの間違い
DATA SEGMENT
CODE ENDS       ; エラー，CODEはDATAの間違い
```

USE32 not allowed without .386

80386プロセッサが指定されていないにもかかわらず，32ビットセグメントが定義されています。32ビットセグメントを定義する場合は，あらかじめ**.386**または**.386P**疑似命令で80386プロセッサを指定してください。

User-generated error

疑似命令によって強制的に発生させたエラーです。

例:

```
.ERR     ; エラー発生疑似命令
```

Value out of range

定数は有効ですが，その命令に対しては大きすぎます。

例:

```
DB  400
```

致命的エラーメッセージ

Turbo Assemblerでは，致命的エラーメッセージが表示されるとアセンブルはただちに中断します。この場合，エラーの原因にかかわらずアセンブル処理は続行不可能となります。以下に致命的エラーメッセージの一覧を示します。

Bad switch ***opt***

無効または不正なコマンドラインが使われています。コマンドラインオプションの詳細は，ユーザーズガイドの第3章を参照してください。

Can't find @file ***fname***

指定された間接コマンドファイルが見つかりません。ファイル名が正しく指定されているかどうか確認してください。ファイル名に対して，自動的にデフォルトの拡張子は付けられません。クロスリファレンスファイルの作成が指定されましたが，ディスクに十分な空き領域がありません。

Can't locate file ***fname***

INCLUDE疑似命令で指定されたファイルが見つかりません。このマニュアルの第3章のINCLUDE疑似命令の説明によって，Turbo Assemblerのインクルードファイルの探索方法を確認してください。

Error writing to listing file

リストファイルの作成が指定されましたが，ディスクに十分な空き領域がありません。

Error writing to object file

オブジェクトファイルの作成が指定されましたが，ディスクに十分な空き領域がありません。

File not found

コマンドラインで指定したソースファイルが見つかりません。ファイル名を正しく入力したかどうかを確認してください。目的のソースファイルがカレントディレクトリ以外のディレクトリにある場合は，ドライブ名やパス名も正しく入力してください。

File was changed or deleted while assembly in progress

Turbo Assemblerがファイルをオープンした後に，ポップアップユーティリティなどの他のプログラムによってそのファイルが更新または削除されました。Turbo Assemblerは，すでにオープンされているファイルを引き続き再オープンすることはできません。

Insufficient memory to process command line

指定したコマンドラインのサイズが64Kバイトを超えています。またはコマンドラインを展開するとユーザメモリのサイズを超えてしまいます。コマンドラインを短縮するか，より大きなユーザメモリを確保した後にTurbo Assemblerを起動し直してください。

Internal error

Turbo Assemblerの通常処理では，このエラーは発生しません。このメッセージが表示された場合は，エラーの原因となったファイルを保存した後，MSAのカスタマーサポートセンターまでその旨お知らせください。

Invalid command line

Turbo Assemblerを起動するために使ったコマンドラインの形式が正しくありません。

例:

```
TASM ,MYFILE
```

この例では，アセンブルの対象となるソースファイルが指定されていません。Turbo Assemblerのコマンドラインの詳細については，**ユーザーズガイド**の第3章を参照してください。

Invalid number after ***opt***

コマンドラインオプション（スイッチ）の後に，有効な数値引数が指定されていません。コマンドラインの詳細については，**ユーザーズガイド**の第3章を参照してください。

Maximum macro expansion size exceeded

マクロを展開した結果，テキストの量が展開領域を超えています。この領域のサイズは最大64Kバイトもあります。したがって，マクロの展開を無限に実行するようなバグがマクロ内ある場合にのみ，このメッセージは表示されます。

Out of hash space

ハッシュテーブルには，プログラム内で定義された各シンボルに対して1つずつエントリが入っています。十分なユーザメモリがある限り，Turbo Assemblerでは最大16, 384個

のシンボルを定義することができます。これより多くのシンボルを使用する場合は，/**KH**コマンドラインオプションを使って，ハッシュテーブルに登録するシンボル数を設定してください。

Out of memory

ユーザメモリが十分にないため，Turbo Assemblerはファイルをアセンブルすることができません。

TSRプログラム(RAM常駐プログラム)がすでにインストールされている場合は，このプログラムをメモリから消去した後に再度アセンブルを実行してください。この場合，メモリを確実に開放するために，システムをリセットしてからTurbo Assemblerを起動してください。

使用メモリのサイズを小さくするための方法として，ソースファイルを数個に分割してからアセンブルする方法があります。また，アセンブルときに必要なメモリが少なくて済むようにプログラムを変更する方法もあります。さらに，シンボル名を短縮する，マクロ内のコメント数を減らす，プログラム内の前方参照の数を減らすなどの方法があります。

Out of string space

シンボル名，ファイル名，前方参照の追跡情報，マクロテキストなどに対して使用できるメモリが足りません。/**KS**コマンドラインオプションを使って，これらの文字列用の領域を確保してください。通常は，ユーザメモリの約半分が文字列領域として使われます。

Too many errors found

多数のエラーが発生したために，アセンブル処理が中止されました。特定のエラーから他のエラーが連鎖的に発生することがあります。たとえば，プログラムで使うシンボルが定義されていない場合は，発生するエラーの種類は未定義シンボルエラーの1種類だけです。しかし，その未定義シンボルを参照しているすべての行に対してエラーメッセージが表示されます。

Turbo Assemblerでは，発生したエラーの総数が100に達したとき点でアセンブル処理が中止されます。

Unexpected end of file (no END directive)

ソースファイルの最後に，**END**疑似命令が指定されていません。ソースファイルの最後には，必らず**END**疑似命令を入れてください。

索　引

.(ピリオド)
Idealモードと
MASMモードの比較 265
.(ピリオド)演算子 25
:(コロン)演算子 27
ローカルシンボルと～ 187
:(コロン)疑似命令 88
;;演算子
マクロ内の～ 76
?記号(疑問符)
～演算子 28
!(感嘆符)
MAKEユーティリティ 303
!演算子
マクロ内の～ 74
(シンボルの下線記号)
Turbo Cと～ 194
/演算子 26
¥(円記号)
メイクファイルのコメント 290
()演算子 20
[](大カッコ)演算子 29
+(二項)演算子 22
+(単項)演算子 22
-(ハイフン)
メイクファイルのプレフィクス 296
-(単項)演算子 24
-(二項)演算子 23
=疑似命令 89
Idealモードと
MASMモードの比較 274
%(パーセント記号)
～疑似命令 78
マクロ内の～ 75
#(シャープ記号) 290
&演算子
マクロ内の～ 72
*演算子 21
@記号(単価記号)
TLINKと～ 315
メイクファイルのプレフィクス 296
ローカルシンボルと～ 187
.186疑似命令 79
.286疑似命令 80
.286C疑似命令 81
.286P疑似命令 81
.287疑似命令 82
2パスアセンブラ
～との互換性 136,166
.386疑似命令 83
.386C疑似命令 83
.386P疑似命令 84
.387疑似命令 85
/3オプション
TLINK 322
80186プロセッサ
.186疑似命令 79,217
80286プロセッサ
.286疑似命令 80,217
.286C疑似命令 81
.286P疑似命令 81
80287コプロセッサ
.287疑似命令 82,218
80386プロセッサ
.386疑似命令 83,218
.386C疑似命令 83
.386P疑似命令 84,218
Idealモードと
MASMモードの比較 280
算術演算 18

80387コプロセッサ
.387疑似命令 85,219
.8086疑似命令 86
8086プロセッサ
.8086疑似命令 86,219
.8087疑似命令 87
8087コプロセッサ
.8087疑似命令 87,219
エミュレーション 204

【A】

ALIGN疑似命令 90
Idealモードと
MASMモードの比較 274
.ALPHA疑似命令 91
AND命令 30
ARG疑似命令 92
BYTE型と～ 93
Turbo Debuggerと～ 92,226
ASSUME疑似命令 95

【B】

%BIN疑似命令 96
BUILTINS.MAK 309
BYTE演算子 31
BYTE型
ARG疑似命令と～ 93

【C】

/Cオプション(TLIB) 333
/cオプション(TLINK) 321
C → Turbo C
CATSTR疑似命令 67,192
.CODE疑似命令 98
@codeシンボル 9
CODESEG疑似命令 99
@CodeSizeシンボル 9
COMMAND.COM
→ MAKEユーティリティと～ 296
COMM疑似命令 100
COMMENT疑似命令 102
%CONDS疑似命令 103
CONST疑似命令 104
.CONST疑似命令 104
@Cpuシンボル 10
CREF 202,262
%CREFALL疑似命令 106
%CREFREF疑似命令 107
%CREFUREF疑似命令 108
%CREF疑似命令 105
.CREF疑似命令 105
CSレジスタ
.CODE疑似命令と～ 98
%CTLS疑似命令 109
@cursegシンボル 11

【D】

/dオプション
TLINK 321
.DATA疑似命令 110
@dataシンボル 11
.DATA?疑似命令 111
DATASEG疑似命令 112
@DataSizeシンボル 12,193
??Dateシンボル 12
DB疑似命令 113
DD疑似命令 114
Turbo Debuggerと～ 114
%DEPTH疑似命令 116
DF疑似命令 117

Turbo Debuggerと～ 117
DISPLAY疑似命令 119
DOS
MAKEユーティリティと～ 297
時刻 15
セグメントの順序付け 120
日付 12
DOSワイルドカード
TLIBユーティリティ 331
DOSSEG疑似命令 120
DP疑似命令 121
DQ疑似命令 121
DT疑似命令 122
DUP演算子 32
DW疑似命令 123
Turbo Debuggerと～ 123
DWORD演算子 33

【E】

/eオプション(TLINK) 321
!elif命令(MAKE) 304
ELSE疑似命令 124
!else命令(MAKE) 304
ELSEIF疑似命令 125
EMUL疑似命令 126
END疑似命令 127
ENDIF疑似命令 128
!endif命令(MAKE) 304
ENDM疑似命令 129
ENDP疑似命令 130
ENDS疑似命令 131
EQ演算子 34
EQU疑似命令 7,132
Idealモードと
MASMモードの比較 274
THIS演算子と～ 63
.ERR疑似命令 133
ERR疑似命令 134
.ERR1疑似命令 135
.ERR2疑似命令 136
.ERRB疑似命令 137
.ERRDEF疑似命令 138
.ERRDIF疑似命令 139
.ERRDIFI疑似命令 140
.ERRE疑似命令 141
.ERRIDN疑似命令 142
.ERRIDNI疑似命令 143
ERRIF疑似命令 144
ERRIF1疑似命令 144
ERRIF2疑似命令 144
ERRIFB疑似命令 145
ERRIFDEF疑似命令 145
ERRIFDIF疑似命令 145
ERRIFDIFI疑似命令 146
ERRIFE疑似命令 146
ERRIFIDN疑似命令 146
ERRIFIDNI疑似命令 147
ERRIFNB疑似命令 147
ERRIFNDEF疑似命令 147
.ERRNB疑似命令 148
.ERRNDEF疑似命令 149
.ERRNZ疑似命令 150
!error命令(MAKE) 306
EVEN疑似命令 151
EVENDATA疑似命令 152
.EXEファイル
TLINK 317
EXITM疑似命令 153
Extended Dictionary(拡張辞書)
321,328

～の作成 335
TLIBのフラグ 330
EXTRN疑似命令 154
Idealモードと
MASMモードの比較 155

【F】

.FARDATA疑似命令 156
FARDATA疑似命令 158
@fardataシンボル 13
@fardata?シンボル 13
.FARDATA?疑似命令 157
FAR演算子 35
farデータ 13
DF疑似命令 117
DP疑似命令 121
FARDATA疑似命令 158
.FARDATA疑似命令 156
.FARDATA?疑似命令 157
UFARDATA疑似命令 255
演算子 35
@FileNameシンボル 14
??Filenameシンボル 14
FWORD演算子 36

【G】

GE演算子 36,37
GLOBAL疑似命令 159
Idealモードと
MASMモードの比較 277
GREPユーティリティ 336-343
演算子 338
検索文字列 338
コマンドラインの構文 336
使用例 339
ファイル指定 339
GROUP疑似命令 161
GT演算子 38

【H】

HIGH演算子 39

【I】

/iオプション
TLINK 320
I/O
画面 216
Idealモード 163
MASMモードとの比較 271—276
インクルードファイル 175
演算子の優先順位 18
定義済みシンボル 7
式の文法 268
セグメントグループ 161
ライブラリのリンク 176
ラベル 181
ローカルシンボル 187
IF疑似命令 164
!if命令(MAKE) 304
IF1疑似命令 165
IF2疑似命令 166
IFB疑似命令 167
IFDEF疑似命令 168
IFDIF疑似命令 169
IFDIFI疑似命令 169
IFE疑似命令 170
IFIDN疑似命令 171
IFIDNI疑似命令 171
IFNB疑似命令 172
IFNDEF疑似命令 173

%INCL疑似命令 174
!include命令(MAKE) 304
INCLUDE疑似命令 175
INCLUDELIB疑似命令 176
INSTR疑似命令 177,192
IRP疑似命令 178
IRPC疑似命令 179

【J】

/JJUMPSオプション 180
JUMPS疑似命令 180

【L】

/lオプション
　TLINK 320
LABEL疑似命令 181
　Idealモード 181
.LALL疑似命令 182
LARGE演算子 40
LE演算子 41
LENGTH演算子 42
.LFCOND疑似命令 182
%LINUM疑似命令 183
.LIST疑似命令 184
%LIST疑似命令 184
LOCAL疑似命令 185
　Turbo Debuggerと～ 186
LOCALS疑似命令 187
LOOP命令
　Idealモードと
　　MASMモードの比較 280
LOW演算子 43
LT演算子 44

【M】

/mオプション
　TLINK 316,318
MACRO疑似命令 189
%MACS疑似命令 190
MAKEユーティリティ 284—314
　BUILTINS.MAKEファイル 309
　TOUCHユーティリティ 309
　エラーメッセージ 310
　大文字小文字の区別 310
　疑似命令 303
　構文 307
　コマンドラインオプション 310
　コンパイル(アセンブル)の強制 309
　使用例 284
　～の中断 308
　マクロ 298
　メイクファイルの作成 289—307
MASK演算子 45
MASM疑似命令 191
MASM51疑似命令 192,275
MASMとの互換性 271-276
　80386プロセッサ 280
　ALIGN疑似命令 274
　BYTE演算子 31
　DWORD演算子 33
　FAR演算子 35
　FWORD演算子 36
　GLOBAL疑似命令 277
　LOOP命令(80386) 280
　NEAR演算子 48
　PROC演算子 52
　Quirksモード 192,231,273,274,276
　QWORD演算子 55

TBYTE演算子 62
Turbo Cと～ 194
UNKNOWN演算子 66
??Versionシンボル 15
イコール定義疑似命令 89,274
演算子の優先順位 18
改良点 277-281
環境変数 272
疑似命令 77,191,192,210
共用体 278
コマンドラインの構文 277
算術演算 275
式の文法 265
シフト 281
ジャンプ 273
条件ジャンプ 278
セグメントアラインメント 274
セグメントオーバライド 279
セグメントグループ 161
セグメントの順序付け 91
セグメントレジスタ 273
代替疑似命令 280
定義済みシンボル 7
定義済み変数 280
定数セグメント 279
バージョン5.1 192,210,271,276
引数の受け渡し 279
符号付き演算命令 275
浮動小数点エミュレーション 279
浮動小数点フォーマット 272
変数の再定義 192
リスト制御 280
ローカルシンボル 208,278
論理演算 275
MASMモード → MASMとの互換性
/MLオプション
大文字小文字の区別 7
MODEL疑似命令 197
.MODEL疑似命令 193
RETURNSキーワードと～ 93,226
MOD演算子 46
MS-DOS → DOS
MULTERRS疑似命令 198

【N】

/Nオプション
OBJXREF 347
TLINK 320
NAME疑似命令 199
NEAR演算子 48
nearデータ 11
：(コロン)疑似命令 57
演算子 48
%NEWPAGE疑似命令 200
NE演算子 47
%NOCONDS疑似命令 201
%NOCREF疑似命令 202
%NOCTLS疑似命令 203
NOEMUL疑似命令 204
%NOINCL疑似命令 205
NOJUMPS疑似命令 206
%NOLIST疑似命令 207
NOLOCALS疑似命令 208
%NOMACS疑似命令 209
NOMASM51疑似命令 210
NOMULTERRS疑似命令 211
%NOSYMS疑似命令 212
NOTHINGキーワード 95
%NOTOC疑似命令 77
%NOTRUNC疑似命令 213

NOT演算子 79
NOWARN疑似命令 214

【O】

OBJXREFユーティリティ 344—356
/Nオプション 347
/Oオプション 347
エラーメッセージ 354
警告 354
構文 344
コマンドラインオプション 345
使用例 353
ディレクトリの変更 347
レポート 346,348
OFFSET演算子 50
OR命令 51
ORG疑似命令 215
%OUT疑似命令 216
/Oオプション
OBJXREF 347

【P】

P186疑似命令 217
P286疑似命令 217
P286N疑似命令 217
P287疑似命令 218
P386疑似命令 218
P386N疑似命令 218
P387疑似命令 219
P8086疑似命令 219
P8087疑似命令 219
PAGE疑似命令 220
%PAGESIZE疑似命令 221
Pascal → Turbo Pascal
PC-DOS → DOS
%PCNT疑似命令 222
PNO87疑似命令 223
%POPLCTL疑似命令 224
PROC演算子 52
PROC疑似命令 225
RETURNSキーワード 94,226
Prolog → Turbo Prolog
PTR演算子 53
PUBLIC疑似命令 228
PURGE疑似命令 229
%PUSHLCTL疑似命令 230
PWORD演算子 54

【Q】

QUIRKS疑似命令 192,231,271,276
Quirksモード 124,271,276
QWORD演算子 55

【R】

.RADIX疑似命令 232
RADIX疑似命令 232
RECORD疑似命令 233
REPT疑似命令 235
RETURNSキーワード 93,226

【S】

/Sオプション 240
/sオプション(TLINK) 316,318
.SALL疑似命令 236
SEG演算子 56
SEGMENT疑似命令 237
.SEQ疑似命令 240
.SFCOND疑似命令 241
SHL演算子 57
SHORT演算子 58

SHR演算子 59
SIZE演算子 60
SIZESTR疑似命令 192,241
SMALL演算子 61
.STACK疑似命令 242
STACK疑似命令 242
STRUC疑似命令 243
Idealモードと
MASMモードの比較 243
UNION(共用体)との比較 256
SUBSTR疑似命令 192,245
%SUBTTL疑似命令 247
SUBTTL疑似命令 246
%SYMS疑似命令 248
SYMTYPE演算子 62

【T】

/tオプション(TLINK) 322
%TABSIZE疑似命令 249
TBYTE演算子 62
TCREFユーティリティ 356-358
TLINK 356
コマンドラインオプション 357
出力 357
%TEXT疑似命令 250
.TFCOND疑似命令 251
THIS演算子 63
??Timeシンボル 15
TITLE疑似命令 252
%TITLE疑似命令 253
TLIBユーティリティ 328-335
拡張辞書 328
拡張辞書フラグ 330
構文 329
使用例 334
操作 331
操作リスト 331
ファイル名 331
レスポンスファイル 333
TLINKユーティリティ 315-327
.COMファイルの作成 322
TCREFと～ 356
拡張辞書 321
エラーメッセージ 333
コマンドラインオプション 318
制約 323
セグメントマップ 318
マップファイル 316,318
レスポンスファイル 317
%TOC疑似命令 77
TOUCHユーティリティ 309
TPASCALメモリモデル 193
%TRUNC疑似命令 254
Turbo C
～とのリンク 194
Turbo Debugger
ARG疑似命令と～ 92,226
DD疑似命令と～ 114
DF疑似命令と～ 117
DW疑似命令と～ 123
LOCAL疑似命令と～ 186
Turbo Link → TLINKユーティリティ
Turbo Pascal
ARG疑似命令と～ 93
.MODEL疑似命令と～ 193
PROC疑似命令と～ 226
～とのリンク 193
Turbo Prolog
～とのリンク 194
Turboライブラリアン

→ TLIBユーティリティ
.TYPE演算子 64
TYPE演算子 65

【U】

UDATASEG疑似命令 254
UFARDATA疑似命令 255
!undef疑似命令 307
UNION疑似命令 243,256
構造体との比較 256
UNKNOWN演算子 66
USES疑似命令 258

【V】

/vオプション(TLINK) 322
??Versionシンボル 15

【W】

WARN疑似命令 260
WIDTH演算子 68
@WordSizeシンボル 16
WORD演算子 69

【X】

/xオプション(TLINK) 318
.XALL疑似命令 261
.XCREF疑似命令 262
.XLIST疑似命令 262
XOR演算子 70

【あ】

アクションシンボル(TLIB) 331
アセンブル
MAKEユーティリティと～ 284-314
条件～ 124
ENDIF疑似命令 128
EXITM疑似命令 153
エラーメッセージ 133
～疑似命令 164-173
リストファイル 183
暗黙のルール(メイクファイル) 292
イコール定義疑似命令 89,132
Idealモードと
MASMモードの比較 132
インクルードファイル
GLOBAL疑似命令と～ 159
Idealモード 175
～のリスト表示 174,205,241
エラーメッセージ 359-390
MAKEユーティリティに関する～ 306,310
OBJXREF 354
TLINK 323
疑似命令に関する～ 133-150,211,214,258
警告～ 361
条件アセンブルに関する～ 133
シンボルに関する～ 138,145,147,149
致命的～ 388
複数の～ 198
～の不表示 214
マクロに関する～ 148
エリアス 7

演算コプロセッサ
→ 数値演算コプロセッサ
演算子 17-76
→ 各演算子の項を参照
EQ演算子 34
GE演算子 37,38
GREP 338
Idealモードと
MASMモードの比較 29
LE演算子 41,44
NE演算子 47
引用符で囲まれた文字列 74
オフセットサイズ 40,61
加算 22,29
繰り返し 32
減算 23
コメント 76
式のサイズ 33,36,53,54,55,62,69
式の評価 75
乗算 21
剰余(余り) 46
除算 26,46
シンボル 62,64,65
セグメントアドレス 56
代入文字列 72
データサイズ 60
データの定義 42
ポインタ 52,58
マクロ 71-76
優先順位 18,20
リテラル文字列 73
レコード 68
論理演算 30,49,51,57,59,70
オーバレイ
TLINKユーティリティ 323
大文字小文字の区別
MAKEユーティリティ 310
TCREF 357
TLIB 333
文字列の比較 139,142
大文字小文字の区別オプション
TLIB 334
オブジェクトファイル
クロスリファレンス 344—356
モジュール名 199
ライブラリ 328—335
オブジェクトモジュール
OBJXREFユーティリティ 344—356
TLIB 328
TLINK 317
クロスリファレンス 356
オプション
→ コマンドラインオプション
オフセット
SMALL演算子 61
サイズ演算子 40
オペランド
THIS演算子 63
メモリ～ 29

【か】

外部シンボル → シンボル
加算演算子 22,29
下線記号
Turbo Cと～ 194
ローカルシンボルと～ 187
型 → データ型
カッコ演算子( ) 20
環境変数
MASMモード 272

感嘆符(!)演算子
　マクロ内の～ 74
簡略セグメント疑似命令 9,11,13
　メモリモデル 193,197
疑似命令 77-262
　farデータ 255
　Idealモード 163
　Idealモードと
　　MASMモードの比較 280
　MAKEユーティリティ 303
　MASMモード 191,192,210,231
　nearデータ 57
　イコール定義 89,132
　インクルードファイル 174,175
　インクルードファイルのリスト表示
　　205
　引用符で囲まれた文字列 119
　エラーメッセージ 133-150,211,
　　214,260
　外部シンボル 154
　カレントセグメント 215
　偽条件 201,241
　共用変数 100
　偶数アドレス 151,152
　繰り返し 178,179,235
　グローバルシンボル 159
　クロスリファレンス 105,202,262
　コードセグメント 99
　構造体/共用体 131,243,256
　コプロセッサのエミュレーション
　　126
　コメント 102
　式 232
　条件アセンブル～ 124,125,128,
　　164-173,182,241,251
条件ジャンプ 180,206
シンボル 138,145,147,187
シンボルテーブル 248
シンボルテーブルの不表示 212
スタック 92
スタックセグメント 242
整数定数 232
セグメント 237,240,254,255
　～アラインメント 90,151
　～グループ 161
　～の順序付け 91,120,240
　～名 95
ダブルワードの定義 114
データサイズ 121,122,123
データセグメント 104,110-112,
　254,255
バイトの定義 113
パブリックシンボル 228
浮動小数点の不表示 223
プログラムの終了 127
プロシージャ 130,225
プロセッサ制御 217-219
プロセッサモード 79-87
ポインタ 117,121
マクロ 189,229,235,236
マクロの展開 153,182,190,209,261
メモリモデル 193,197
モジュール名 199
文字列の位置 177
文字列のサイズ 241
文字列の定義 245
文字列の連結 97
ライブラリのリンク 176
ラベル 181
リストの制御 203

リストファイル 103,105-109,174,
182,183,200,207,213,221-222,
224,230,236,241,246-253,262
レコード 233
レジスタのプッシュ/ポップ 258
ローカルシンボル 187,208
ローカル変数 185
疑問符(?)
～演算子 28
～を使ったシンボル 8
共用体
Idealモードと
MASMモードの比較 278
疑似命令 256
クオッドワード
DQ疑似命令 121
演算子 55
繰り返し命令 32,178,179,235
グローバルシンボル 159
クロスリファレンス 356—358
クロスリファレンス
TCREFユーティリティ 356-358
～を無効にする 202
リストファイル中の～ 105,107,
202,262
非参照シンボル 107
クロスリファレンスユーティリティ
→ OBJXREFユーティリティ
→ TCREFユーティリティ
警告メッセージ 361
TLINK 323
疑似命令 260
～の不表示 214
減算
Idealモードと
MASMモードの比較 23
演算子 23
コードセグメント
～疑似命令 99
語意に関する文法 263
構造体
ENDS疑似命令 131
LABEL疑似命令と～ 181
疑似命令 243
～の定義 243
～のネスト 243
ピリオド演算子 25
構文 263-270
語意に関する文法 263
コマンドラインの～
→ コマンドラインの構文
構文のまとめ 263-270
コマンドラインオプション
MAKEユーティリティ 310
TLINK 318
コマンドラインの構文
GREP 336
Idealモードと
MASMモードの比較 277
MAKEユーティリティ 307
OBJXREF 344
TLIB 329
コマンドリスト(メイクファイル) 295
コメント(注釈) 102
～の不表示 76
メイクファイル 290
コロン演算子(:) 27
ローカルシンボルと～ 187
コロン疑似命令(:) 88
コンパイラオプション

→ 各オプションを参照

【さ】

サイズ → データ
算術演算 18
Idealモードと
MASMモードの比較 275
式
10バイトサイズ 62
farポインタサイズ 36,54
～内の演算子 17-70
クオッドワードサイズ 55
～のサイズ 53
整数定数 232
ダブルワードサイズ 33
バイトサイズ 31
～の評価 75
評価の順序 18
ワードサイズ 69
時刻 15
シフト
Idealモードと
MASMモードの比較 281
SHL演算子 57
SHR演算子 59
ジャンプ
Idealモードと
MASMモードの比較 273
Quirksモード 273
SHORT演算子と～ 58
サイズ 180
条件～ 180
Idealモードと
MASMモードの比較 184
サイズ 206
前方参照～ 180
終了
END疑似命令と～ 127
条件アセンブル
ELSE疑似命令 124
ELSEIF疑似命令 125
ENDIF疑似命令 128
EXITM疑似命令 153
IF疑似命令 164
IF1疑似命令 165
IF2疑似命令 166
IFB疑似命令 167
IFDEF疑似命令 168
IFDIF疑似命令 169
IFE疑似命令 170
IFIDN疑似命令 171
IFNB疑似命令 172
IFNDEF疑似命令 173
エラーメッセージ 133
画面表示 216
偽条件 201,241
マクロ 169,171,172
リストファイル 183,201,241,251
条件疑似命令(MAKE) 304
条件ジャンプ → ジャンプ
乗算
演算子 21
剰余演算子 46
除算
演算子 26,46
シンボル 7-16
@code 9
@CodeSize 9
@Cpu 10
@curseg 11

@data 11
@DataSize 12
??Date 12
@fardata 13
@FileName 14
??Filename 14
SYMTYPE演算子 62
??Time 15
.TYPE演算子 64
??Version 15
@WordSize 16
エラーメッセージ 138,145,147,149
エリアス 7
演算子 65
外部～
EXTRN疑似命令 154
共用～ 100
Idealモードと
MASMモードの比較 100-101
複数の～ 100
グローバル～ 159
クロスリファレンス 356-358
クロスリファレンス 106
～の定義 181
パブリック～ 228
非参照～ 107
未定義～ 149
ローカル～ 187
～を無効にする 208
シンボルテーブル
リストファイル 248
～を無効にする 212
数値演算コプロセッサ
アセンブルを無効にする 223
エミュレーション 126,204
スタック
ARG疑似命令 92
スタックセグメント
疑似命令 242
スタックセグメント疑似命令 242
スラッシュ(/)演算子 26
整数
～定数 232
セグメント
ASSUME疑似命令 95
NOTHINGキーワード 95
OFFSET演算子 50
アドレス演算子 56
～アラインメント
Idealモードと
MASMモードの比較 270
～型 237
アルファベット～オーダ 91
～オーバライド 27
Idealモードと
MASMモードの比較 279
～の終わり 131
カレント～ 11,215
～疑似命令 237
OFFSET演算子 50
簡略～ 9,11,13
メモリ モデル 193,197
グループ 11,161
Idealモード 161
～結合型 238
～サイズ 16
シーケンシャル～オーダ 240
～の順序付け 120,240
スタック～ 242
データ～ 254

～の定義 237
定数
Idealモードと
MASMモードの比較 279
～名 95
～レジスタ
→ 各レジスタの項を参照
Idealモードと
MASMモードの比較 273
Quirksモード 273
セグメントのグループ化 11,161
セミコロン(;)
マクロ内の～ 76
前方参照
FAR演算子 35

【た】

大カッコ([]) 29
大カッコ演算子([]) 29
他のアセンブラとの互換性 91,271-276
→ MASMとの互換性
ダブルワード
DD疑似命令 114
DWORD演算子 33
致命的エラーメッセージ 388
定義済みシンボル → シンボル
データ
～型
UNKNOWN 66
～サイズ
DQ疑似命令 121
DT疑似命令 122
SIZE演算子 60
初期化されない～ 111
～の定義 28,32,42
データ構造 → 構造体
データセグメント
EVENDATA疑似命令 152
疑似命令 104,110,111
初期化されない～ 254
データ転送 → データ
データの複写 → データ
定数
整数 232
セグメント
Idealモードと
MASMモードの比較 279
ディスクオペレーティングシステム
→ DOS
テキスト → 文字列
デバッグ 92,114,117,123,186,226
マップファイルと～ 318
等号(=) 89
Idealモードと
MASMモードの比較 274

【は】

バージョン番号(Turbo Assembler) 15
パーセント記号(%)
疑似命令 78
マクロ内の～ 75
バイト
DB疑似命令 113
パイプ 297
パブリックシンボル 228
パブリック名
OBJXREFユーティリティ 344,349
汎用レジスタ
→ 各汎用レジスタの項を参照
大文字小文字の区別と～ 139,142

引数の受け渡し
Idealモードと
MASMモードの比較 279
日付 12
ビット単位のOR演算子 51
ビットごとの補数をとる演算子 49
ビットフィールド
疑似命令 233
ビットマスク 45
ピリオド
Idealモードと
MASMモードの比較 265
～演算子
Idealモード 25
ファイル
MAKEユーティリティ 284-314
アセンブル 14
～のインクルード(MAKE) 303
オブジェクト～ 328-335
OBJXREFユーティリティ
344-355
～の管理(MAKE) 284-314
再コンパイル(アセンブル)の
強制 309
～の命名
TLIB 331
～内の文字列検索(GREP) 336-343
～のリスト表示 → リストファイル
レスポンス～ 346
レスポンス～(OBJXREF) 347
ファイル名 14
符号付き演算命令
Idealモードと
MASMモードの比較 275
符号の変更 24
不等号カッコ(＜＞)
マクロ内の～ 73
マクロ内の～演算子 73
浮動小数点
Idealモードと
MASMモードの比較 272
～エミュレーション 204
Idealモードと
MASMモードの比較 279
コプロセッサ命令のアセンブルを
無効にする 223
～命令 1
プレフィクス
メイクファイル 296
プログラム管理(MAKE) 284-314
プログラムの終了
END疑似命令と～ 127
プロシージャ
～の開始 225
～の終了 130
ローカル変数 185
プロセッサ型の決定 10
プロセッサ制御疑似命令 217-219
変数
共用～ 100
グローバル～(OBJXREF) 344
定義済み～
Idealモードと
MASMモードの比較 280
～の再定義 192
ローカル～ 185
ポインタ
32ビット～ 121
48ビット～ 117
DF疑似命令 117

PROC演算子 52

【ま】

マイクロソフトアセンブラ
→ MASMとの互換性
マクロ命令
IRP疑似命令と～ 179
IRPC疑似命令と～ 179
MAKEユーティリティ 202
エラーメッセージ 138-143,149-150
～内の演算子 71-76
～の削除 229
条件アセンブル 171
条件アセンブル疑似命令 168,169,172
～の定義 189
～の展開
EXITM疑似命令 153
疑似命令 208
リストファイル 190,261
リストの不表示 236
～のリスト表示 182,224
ローカル変数 185
マップファイル
TLINK 316,318
マニュアルの表記法 3
無条件ジャンプ → ジャンプ
明示的ルール(メイクファイル) 290
命令セット → 各命令の項を参照
メモリ
オペランド 29
～モデル
Turbo Pascal 195
疑似命令 193,199
ポインタ 9,12
文字
引用符で囲んだ～ 74
～の表示 216
リテラル 73
文字の表示 → 文字
モジュール化プログラミング
COMM疑似命令と～ 100
EXTRN疑似命令 154
PUBLIC疑似命令 228
クロスリファレンス 356-358
結合型 1237-238
コードセグメント 99
モジュール名 199
文字列
DISPLAY疑似命令 119
～の位置 177
引用符で囲まれた 74
疑似命令 245
～のサイズ 241
～の定義 97,245
～の表示 119
リテラル 73
～の連結 97
文字列の検索(GREP) 336-343

【や】

ユーティリティ 284-358
GREP 336-343
MAKE 284-314
OBJXREF 344-355
TCREF 356-358
TLIB 328-335
TLINK 315-327
優先順位(演算子) 18,20

【ら】

ライブラリ
　～のインクルード 176
　オブジェクトモジュール 328-335,
　　344-355
　拡張辞書の作成 355
ラベル
　PUBLIC疑似命令と～ 181
　～の定義 181
リストファイル 183
　%BIN疑似命令 96
　～内のインクルードファイル
　　174,205
　エラーメッセージ 135-136
　疑似命令 78,116
　～内の偽条件 201,241
　クロスリファレンス情報 105-107,
　　202,262
　～内のクロスリファレンス情報
　　355,358
　条件アセンブル 251
　条件ブロック 183
　タイトル 246,252,253
　非参照シンボル 107
　マクロの展開 182,190,209
　マクロの不表示 236
　リスト形式 183,200,213,220,
　　221,246-254
　リスト制御疑似命令 103,109,203,
　　224,230,280
　リストの不表示 207,262
リダイレクト 297
リンク 227
　→ TLINKユーティリティ
　Turbo Pascal 193
　高級言語 193
　ライブラリ 176
ルール(メイクファイル) 290-295
レコード
　WIDTH演算子 68
　ビットフィールド 233
　ビットマスク 45
レジスタ → 各レジスタの項を参照
レスポンスファイル
　OBJXREF 345,347
　TCREF 356
　TLIB 333
　TLINK 317
レポート(OBJXREF)
　クラス型による～ 346
ローカル シンボル 187
　Idealモードと
　　MASMモードの比較 189,278
　～を無効にする 208
ローカル変数 185
論理演算
　AND 30
　Idealモードと
　　MASMモードの比較 275
　NOT 49
　OR 51
　SHL 57
　SHR 59
　XOR 70

【わ】

ワード
　DW疑似命令 123
　WORD演算子 69

ワイルドカード → DOSワイルドカード

Turbo Assembler 1.0 リファレンスガイド

1989年 4月 1日　初版発行

著者　**Borland International**
編訳・発行　**(株)ボーランド ジャパン**
〒107 東京都港区南青山7-8-1 小田急南青山ビル
電話 (03)486-1400(代表)

製品に関するお問い合わせは,
下記代理店までお願い致します。

(株)マイクロソフトウェア アソシエイツ
〒107 東京都港区南青山7-8-1 小田急南青山ビル
電話 (03)486-1411(代表)
(03)486-1403(サポートセンター)
FAX (03)486-8905

Printed in Japan

落丁,乱丁はお取り替えいたします。

REFERENCE
GUIDE

TURBO ASSEMBLER®

BORLAND